Introduction to Oracle: SQL and PL/SQL

Student Guide • Volume 2

41010GC11
Production 1.1
April 1998
M06586

Authors

Neena Kochhar
Eileen Lad

Technical Contributors and Reviewers

Christian Bauwens
David Bogdonoff
Gunnar Bohrs
Jacquelyn Bruce
Deatrice Carger
Alice Chang
Larry Cross
Donald De Boer
Gillian Elias
Tushar Gadhia
Pascal Gibert
Fred Gomez
Ellen Gravina
Denise Hurlburt
Kuljit Jassar
Bryan Roberts
Harry Siegersma
Vijayanandan Venkatachalam

Publishers

Jerry Brosnan
Stephanie Jones
Kelly Lee
Lisa Patterson

Contents

13

Other Database Objects

Objectives

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

- **Describe some database objects and their uses**
- **Create, maintain, and use sequences**
- **Create and maintain indexes**
- **Create private and public synonyms**

13-2

Lesson Aim

In this lesson, you will learn how to create and maintain some of the other commonly used database objects. These objects include sequences, indexes, and synonyms.

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	Alternative name for an object

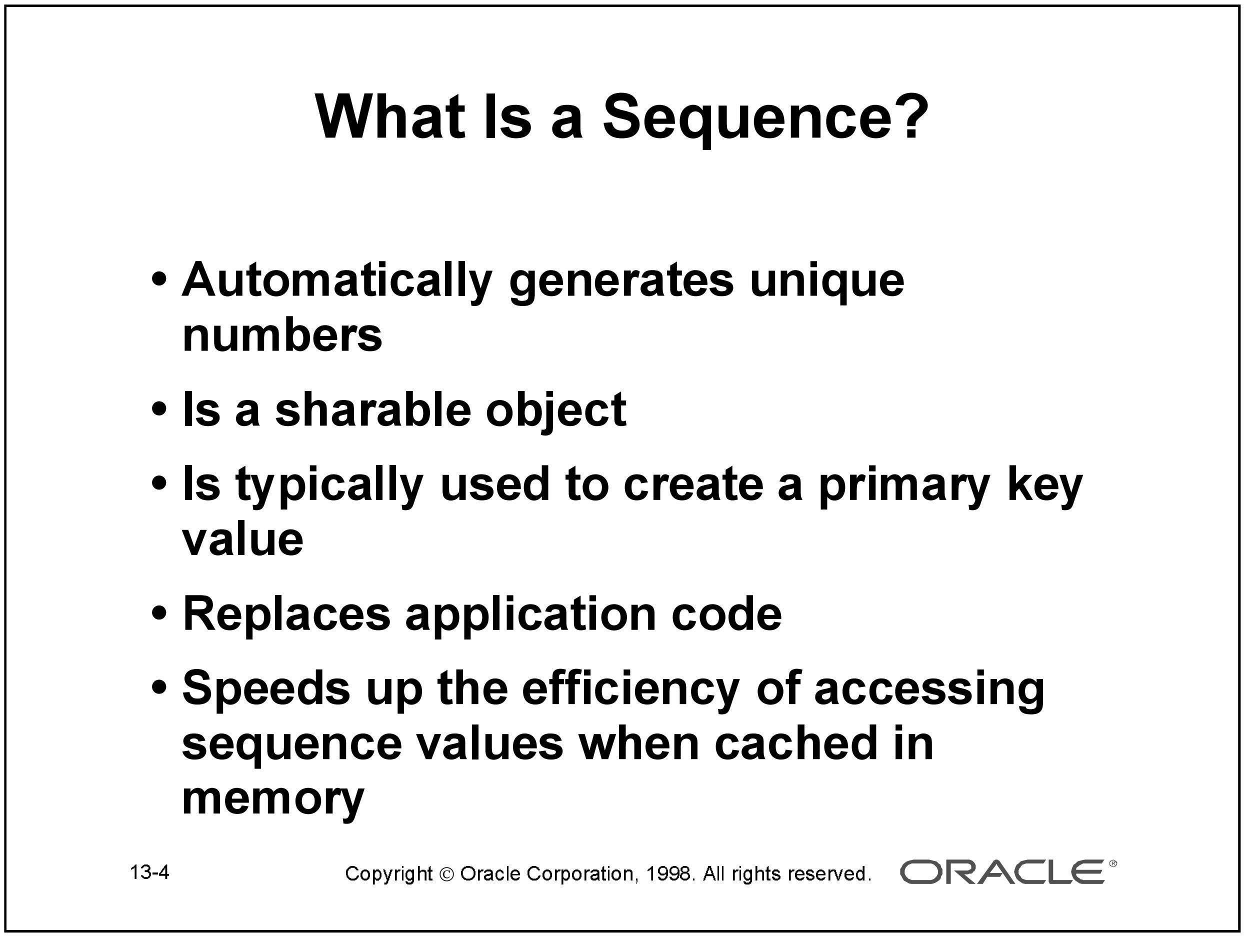
13-3

Database Objects

Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of some queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.



What Is a Sequence?

- Automatically generates unique numbers
- Is a sharable object
- Is typically used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

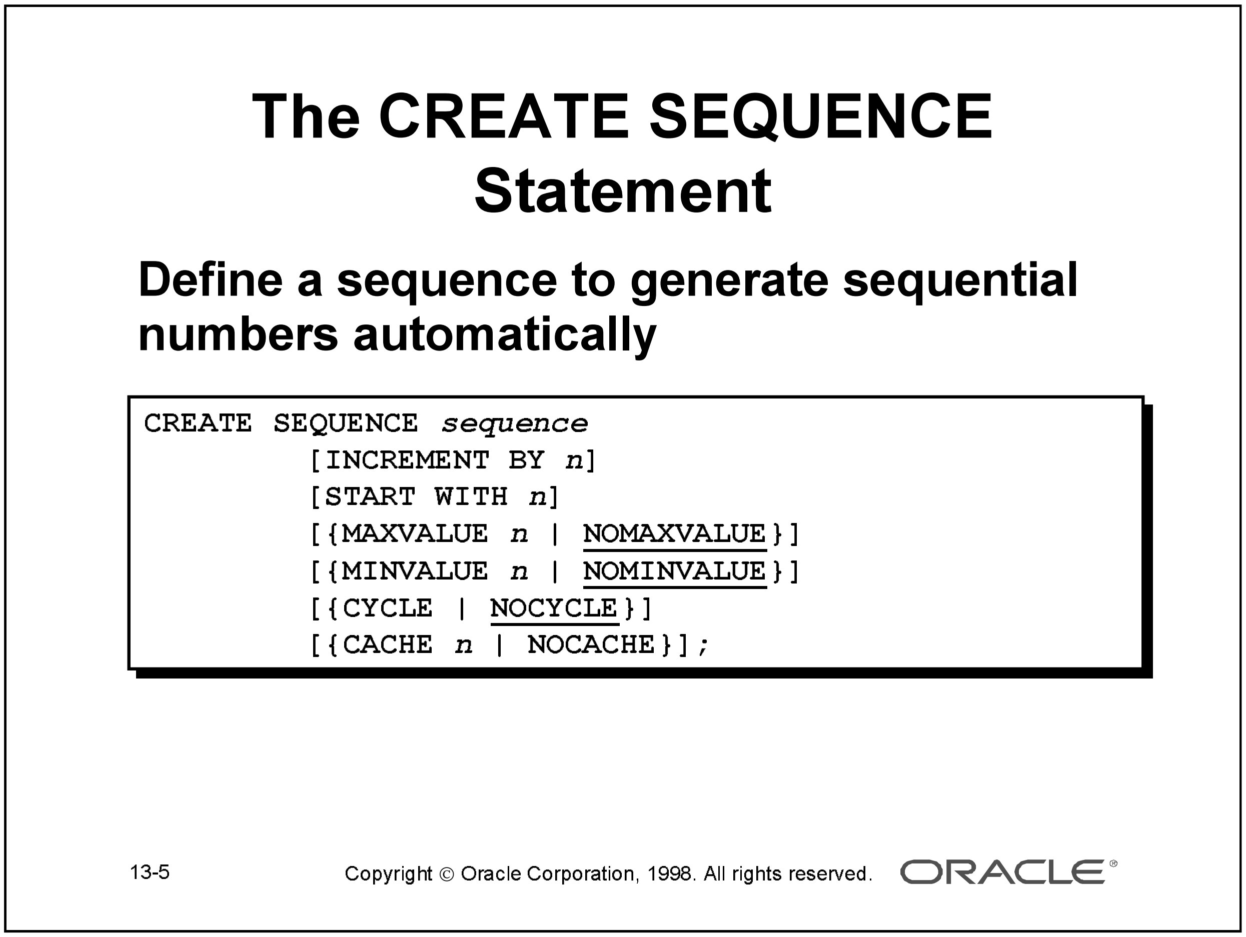
13-4

What Is a Sequence?

A sequence generator can be used to automatically generate sequence numbers for rows in tables. A sequence is a database object created by a user and can be shared by multiple users.

A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle8 routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence generating routine.

Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.



The CREATE SEQUENCE Statement

Define a sequence to generate sequential numbers automatically

```
CREATE SEQUENCE sequence
         [INCREMENT BY n]
         [START WITH n]
         [{MAXVALUE n | NOMAXVALUE}]
         [{MINVALUE n | NOMINVALUE}]
         [{CYCLE | NOCYCLE}]
         [{CACHE n | NOCACHE}];
```

13-5 ORACLE®

Creating a Sequence

Automatically generate sequential numbers by using the CREATE SEQUENCE statement.

In the syntax:

sequence	is the name of the sequence generator.
INCREMENT BY *n*	specifies the interval between sequence numbers where *n* is an integer. If this clause is omitted, the sequence will increment by 1.
START WITH *n*	specifies the first sequence number to be generated. If this clause is omitted, the sequence will start with 1.
MAXVALUE *n*	specifies the maximum value the sequence can generate.
NOMAXVALUE	specifies a maximum value of 10^27 for an ascending sequence and -1 for a descending sequence. This is the default option.
MINVALUE *n*	specifies the minimum sequence value.
NOMINVALUE	specifies a minimum value of 1 for an ascending sequence and -(10^26) for a descending sequence. This is the default option.
CYCLE | NOCYCLE	specifies that the sequence continues to generate values after reaching either its maximum or minimum value or does not generate additional values. NOCYCLE is the default option.
CACHE *n* | NOCACHE	specifies how many values the Oracle Server will preallocate and keep in memory. By default, the Oracle Server will cache 20 values.

Creating a Sequence

- **Create a sequence named DEPT_DEPTNO to be used for the primary key of the DEPT table.**
- **Do not use the CYCLE option.**

```
SQL> CREATE SEQUENCE dept_deptno
  2      INCREMENT BY 1
  3      START WITH 91
  4      MAXVALUE 100
  5      NOCACHE
  6      NOCYCLE;
Sequence created.
```

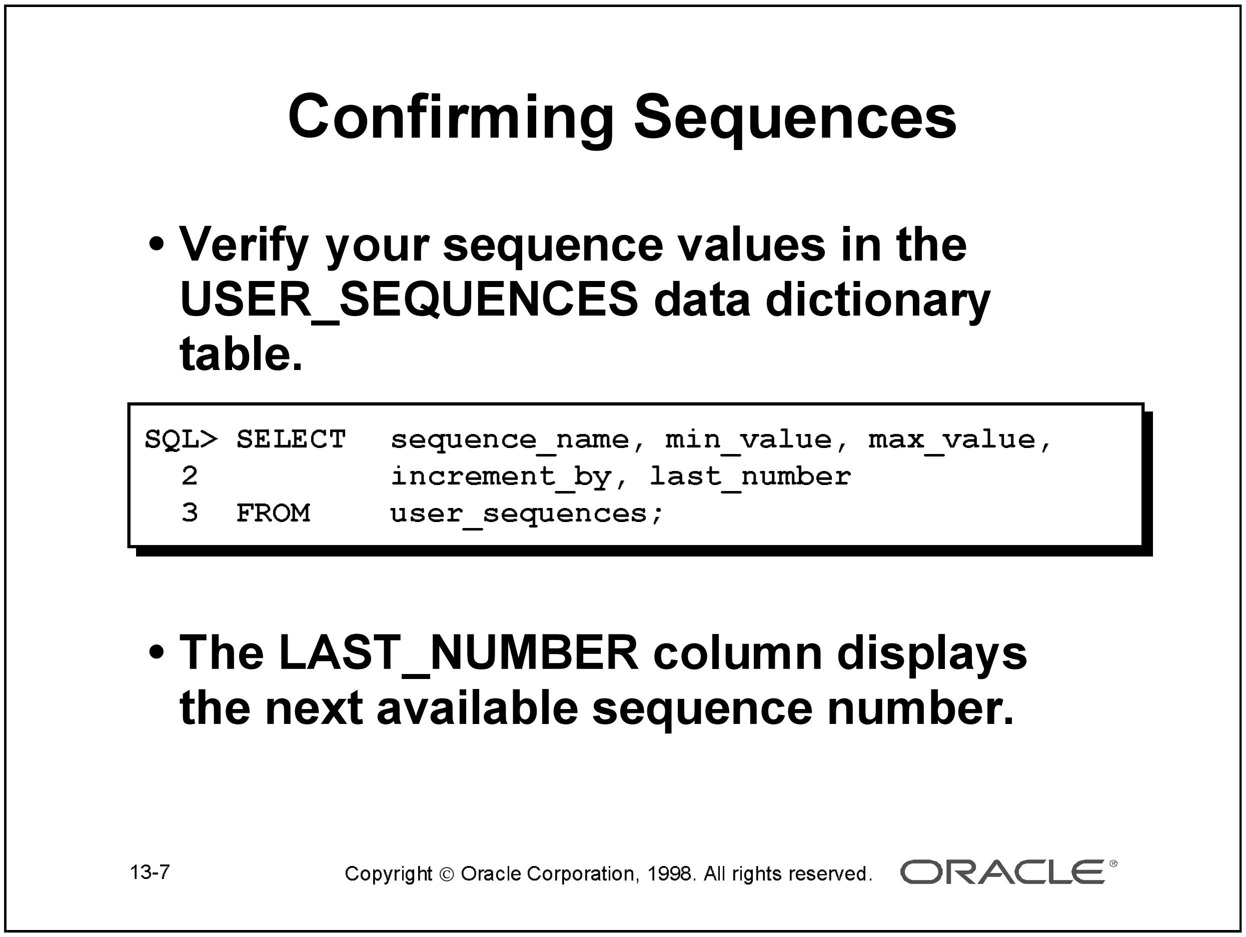
13-6

Creating a Sequence (continued)

The example above creates a sequence named DEPT_DEPTNO to be used for the DEPTNO column of the DEPT table. The sequence starts at 91, does not allow caching, and does not allow the sequence to cycle.

Do not use the CYCLE option if the sequence is used to generate primary key values unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

For more information, see
Oracle Server SQL Reference, Release 8.0, “CREATE SEQUENCE.”



Confirming Sequences

- Verify your sequence values in the USER_SEQUENCES data dictionary table.

```
SQL> SELECT   sequence_name, min_value, max_value,
  2           increment_by, last_number
  3  FROM     user_sequences;
```

- The LAST_NUMBER column displays the next available sequence number.

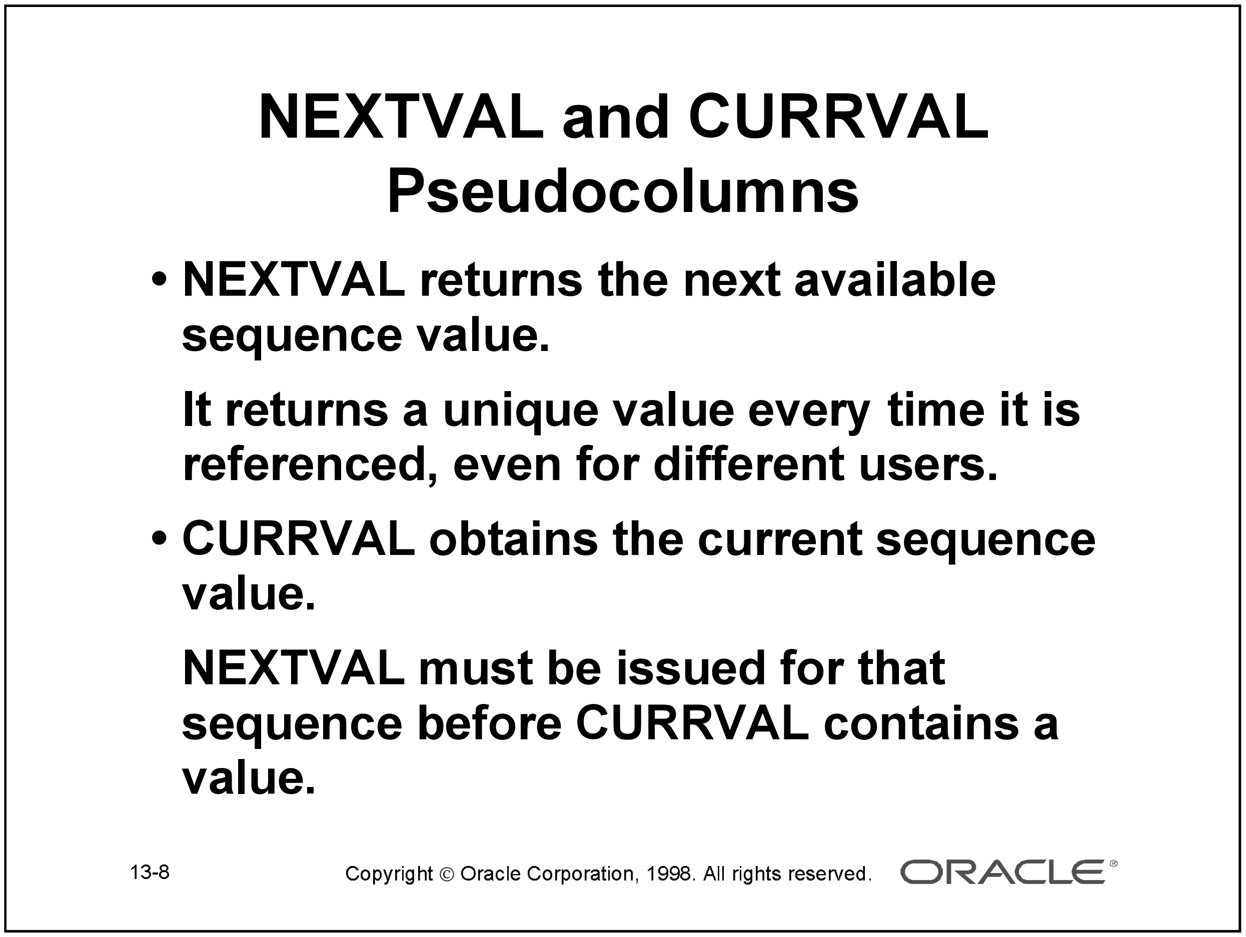
13-7 ORACLE®

Confirming Sequences

Once you have created your sequence, it is documented in the data dictionary. Since a sequence is a database object, you can identify it in the USER_OBJECTS data dictionary table.

You can also confirm the settings of the sequence by selecting from the data dictionary's USER_SEQUENCES table.

```
SEQUENCE_NAME     MIN_VALUE MAX_VALUE INCREMENT_BY LAST_NUMBER
-------------- ----------- --------- ------------ -----------
CUSTID                   1 1.000E+27            1         109
DEPT_DEPTNO              1       100            1          91
ORDID                    1 1.000E+27            1         622
PRODID                   1 1.000E+27            1      200381
```



NEXTVAL and CURRVAL Pseudocolumns

- **NEXTVAL returns the next available sequence value.**

  It returns a unique value every time it is referenced, even for different users.
- **CURRVAL obtains the current sequence value.**

  NEXTVAL must be issued for that sequence before CURRVAL contains a value.

Using a Sequence

Once you create your sequence, you can use the sequence to generate sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

NEXTVAL and CURRVAL Pseudocolumns

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When *sequence*.CURRVAL is referenced, the last value returned to that user's process is displayed.

NEXTVAL and CURRVAL Pseudocolumns

- **NEXTVAL returns the next available sequence value.**

  It returns a unique value every time it is referenced, even for different users.
- **CURRVAL obtains the current sequence value.**

  NEXTVAL must be issued for that sequence before CURRVAL contains a value.

13-9 ORACLE®

Rules for Using NEXTVAL and CURRVAL

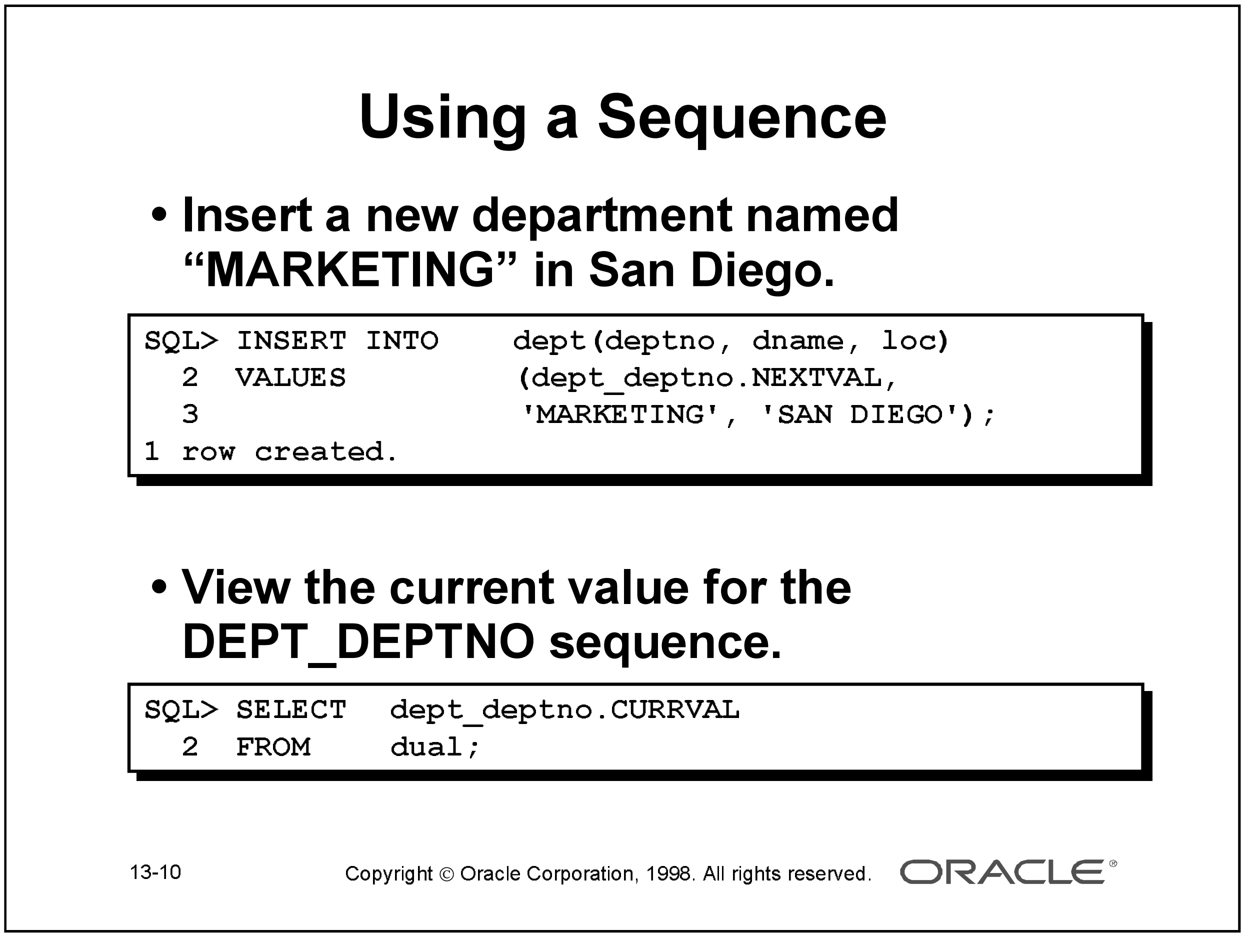
You can use NEXTVAL and CURRVAL in the following:

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You cannot use NEXTVAL and CURRVAL in the following:

- A SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with the GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- A DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

For more information, see
Oracle Server SQL Reference, Release 8.0, “Pseudocolumns” section and “CREATE SEQUENCE.”



Using a Sequence

- Insert a new department named “MARKETING” in San Diego.

```
SQL> INSERT INTO    dept(deptno, dname, loc)
  2  VALUES         (dept_deptno.NEXTVAL,
  3                 'MARKETING', 'SAN DIEGO');
1 row created.
```

- View the current value for the DEPT_DEPTNO sequence.

```
SQL> SELECT   dept_deptno.CURRVAL
  2  FROM     dual;
```

13-10

Using a Sequence

The example on the slide inserts a new department in the DEPT table. It uses the DEPT_DEPTNO sequence for generating a new department number.

You can view the current value of the sequence:

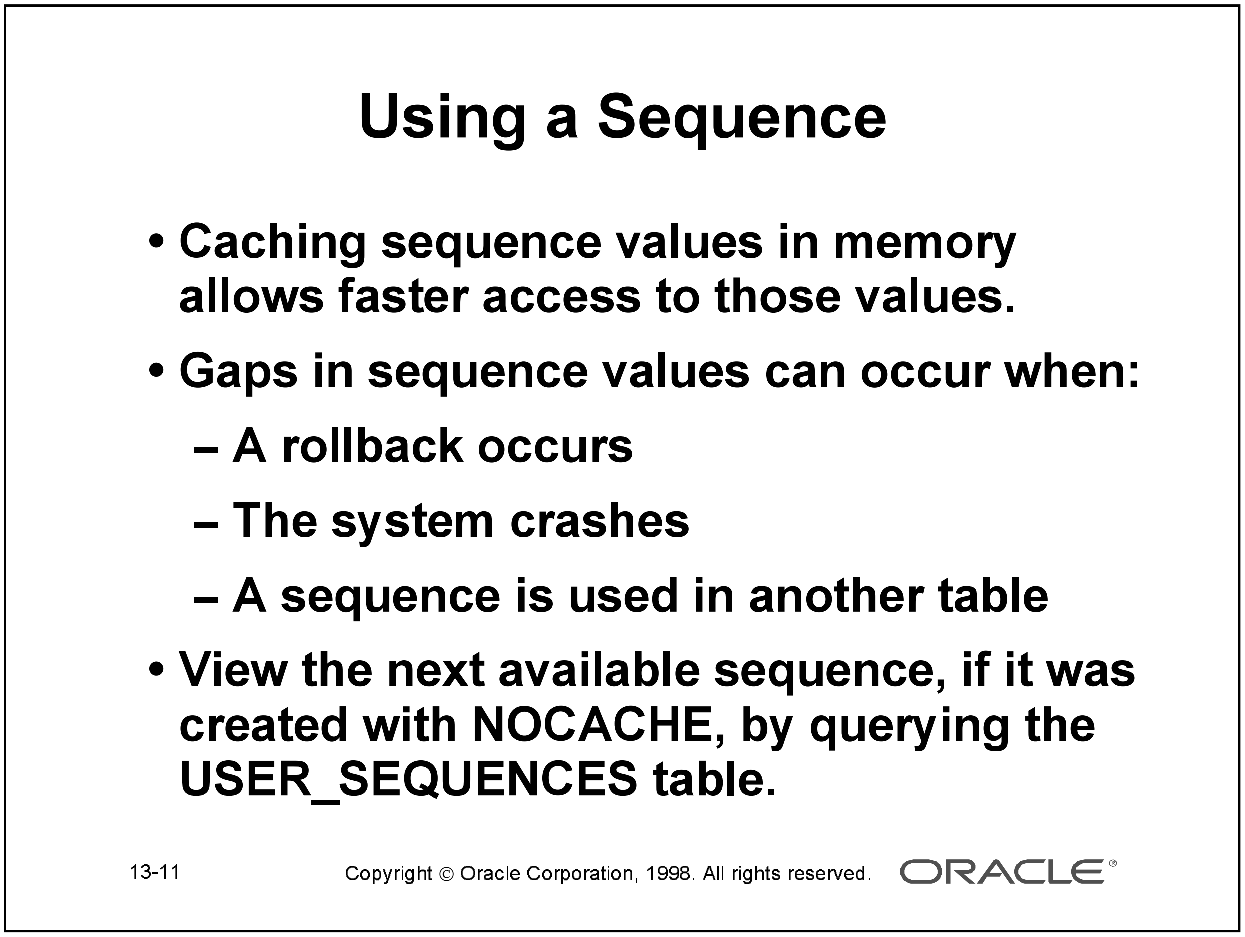
```
SQL> SELECT   dept_deptno.CURRVAL
  2  FROM     dual;
```

```
CURRVAL
-------
     91
```

Suppose now you want to hire employees to staff the new department. The INSERT statement that can be executed repeatedly for all the new employees can include the following code:

```
SQL> INSERT INTO emp ...
  2  VALUES (emp_empno.NEXTVAL, dept_deptno.CURRVAL, ...
```

Note: The above example assumes that a sequence EMP_EMPNO has already been created for generating a new employee number.



Using a Sequence

- Caching sequence values in memory allows faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table
- View the next available sequence, if it was created with NOCACHE, by querying the USER_SEQUENCES table.

13-11 ORACLE®

Caching Sequence Values

Cache sequences in the memory to allow faster access to those sequence values. The cache is populated at the first reference to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence is used, the next request for the sequence pulls another cache of sequences into memory.

Beware of Gaps in Your Sequence

Although sequence generators issue sequential numbers without gaps, this action occurs independent of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.

Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in the memory, then those values are lost if the system crashes.

Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. If this occurs, each table can contain gaps in the sequential numbers.

Viewing the Next Available Sequence Value Without Incrementing It

It is possible to view the next available sequence value without incrementing it, only if the sequence was created with NOCACHE, by querying the USER_SEQUENCES table.

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option.

```
SQL> ALTER SEQUENCE dept_deptno
  2        INCREMENT BY 1
  3        MAXVALUE 999999
  4        NOCACHE
  5        NOCYCLE;
Sequence altered.
```

13-12 ORACLE®

Altering a Sequence

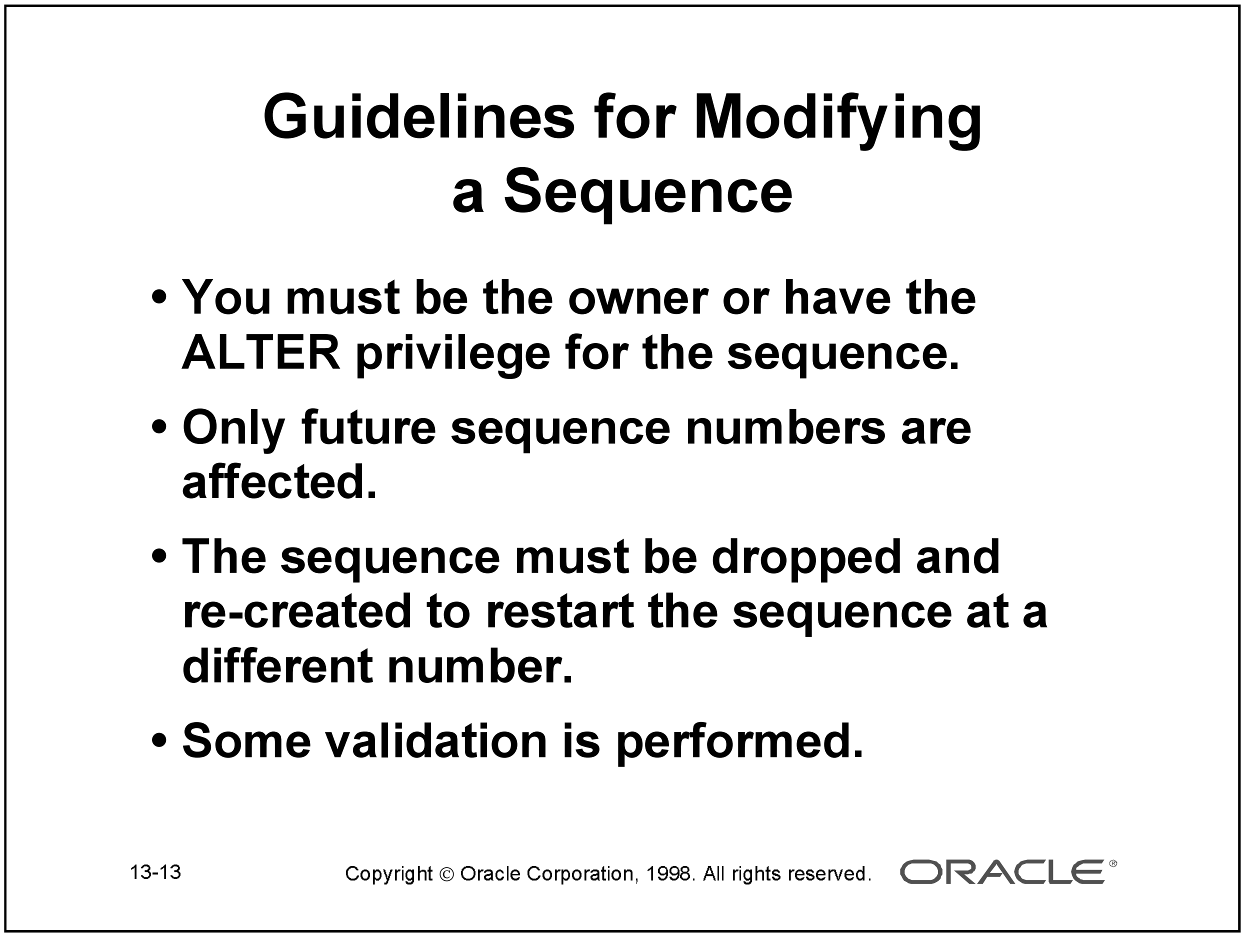
If you reach the MAXVALUE limit for your sequence, no additional values from the sequence will be allocated and you will receive an error indicating that the sequence exceeds the MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.

Syntax

```
ALTER  SEQUENCE  sequence
       [INCREMENT BY n]
       [{MAXVALUE n | NOMAXVALUE}]
       [{MINVALUE n | NOMINVALUE}]
       [{CYCLE | NOCYCLE}]
       [{CACHE n | NOCACHE}];
```

where: *sequence* is the name of the sequence generator.

For more information, see
Oracle Server SQL Reference, Release 8.0, "ALTER SEQUENCE."



Guidelines for Modifying a Sequence

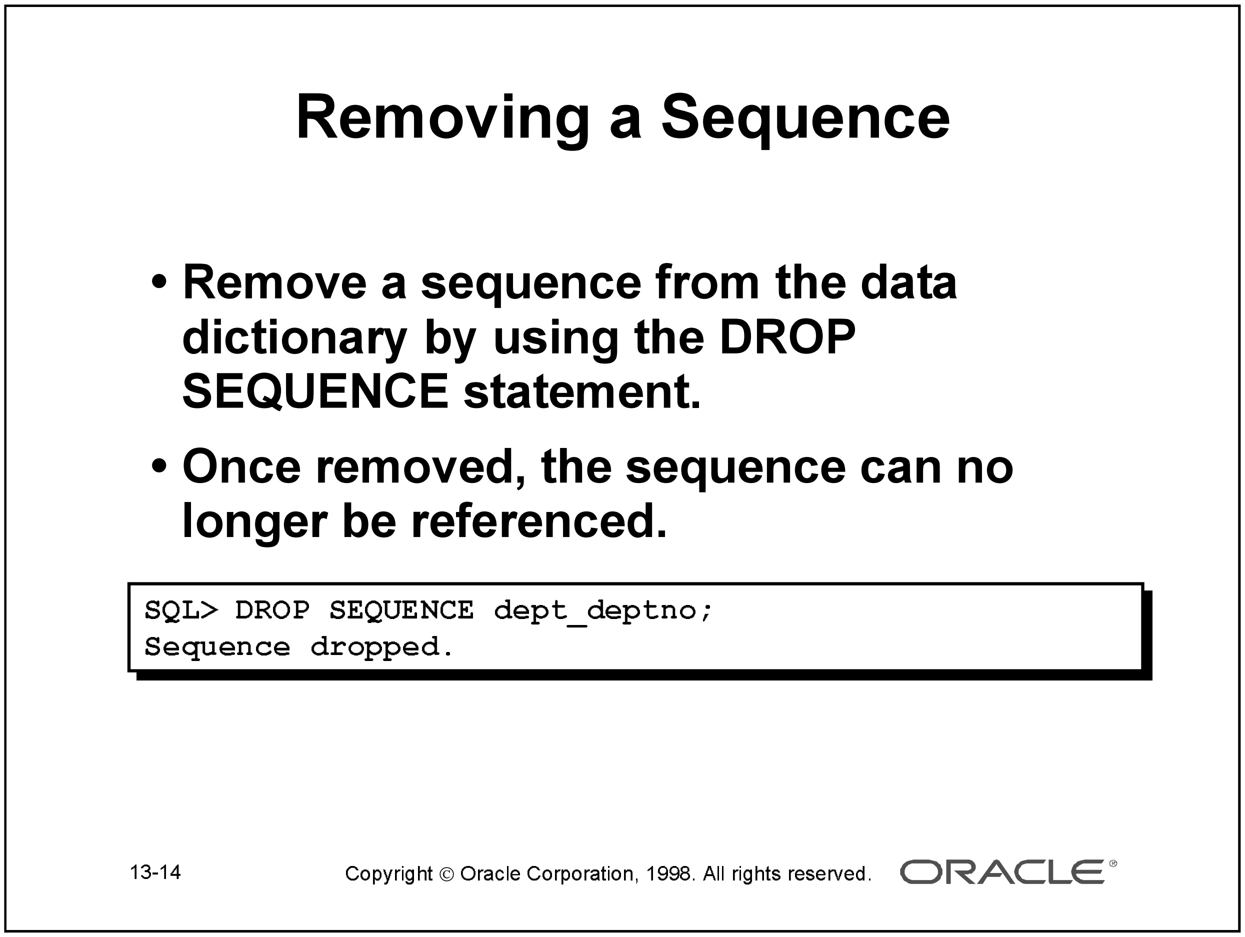
- You must be the owner or have the ALTER privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.

13-13

Guidelines

- You must own or you have the ALTER privilege for the sequence in order to modify it.
- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created in order to restart the sequence at a different number.
- Some validation is performed. For example, a new MAXVALUE cannot be imposed that is less than the current sequence number.

```
 SQL> ALTER SEQUENCE dept_deptno
   2        INCREMENT BY 1
   3        MAXVALUE 90
   4        NOCACHE
   5        NOCYCLE;
ALTER SEQUENCE dept_deptno
*
ERROR at line 1:
ORA-04009: MAXVALUE cannot be made to be less than the current
value
```



Removing a Sequence

- Remove a sequence from the data dictionary by using the DROP SEQUENCE statement.
- Once removed, the sequence can no longer be referenced.

```
SQL> DROP SEQUENCE dept_deptno;
Sequence dropped.
```

13-14

Removing a Sequence

To remove a sequence from the data dictionary, use the DROP SEQUENCE statement. You must be the owner of the sequence or have the DROP ANY SEQUENCE privilege to remove it.

Syntax

```
DROP    SEQUENCE     sequence;
```

where:	*sequence*	is the name of the sequence generator.

For more information, see
Oracle Server SQL Reference, Release 8.0, "DROP SEQUENCE."

What Is an Index?

- Schema object
- Used by the Oracle Server to speed up the retrieval of rows by using a pointer
- Reduces disk I/O by using rapid path access method to locate the data quickly
- Independent of the table it indexes
- Automatically used and maintained by the Oracle Server

13-15

What Is an Index?

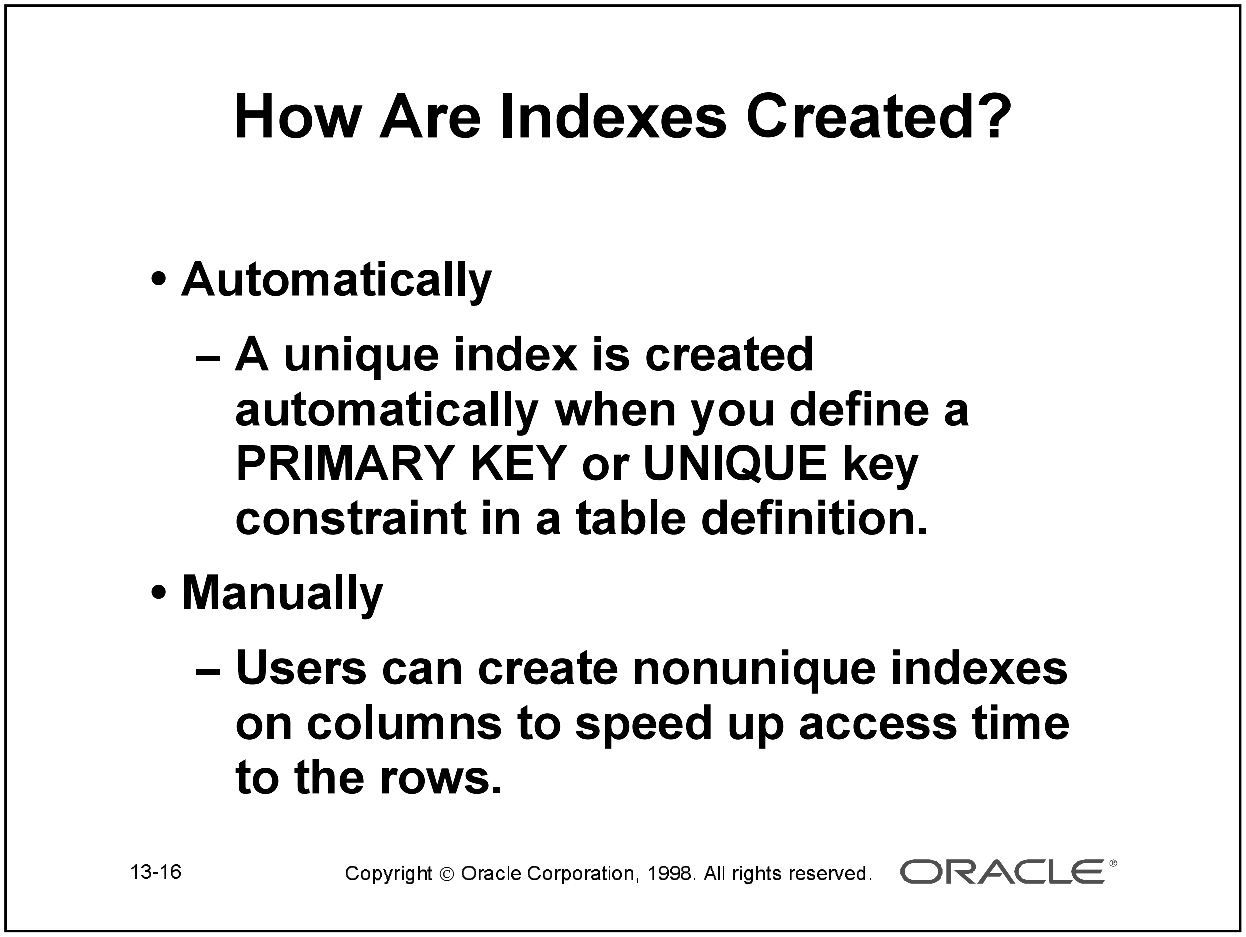
An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. If you do not have an index on the column, then a full table scan will occur.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is automatically used and maintained by the Oracle Server. Once an index is created, no direct activity is required by the user.

Indexes are logically and physically independent of the table they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

Note: When you drop a table, corresponding indexes are also dropped.

For more information, see
Oracle Server Concepts Manual, Release 8.0, "Schema Objects" section, "Indexes" topic.



How Are Indexes Created?

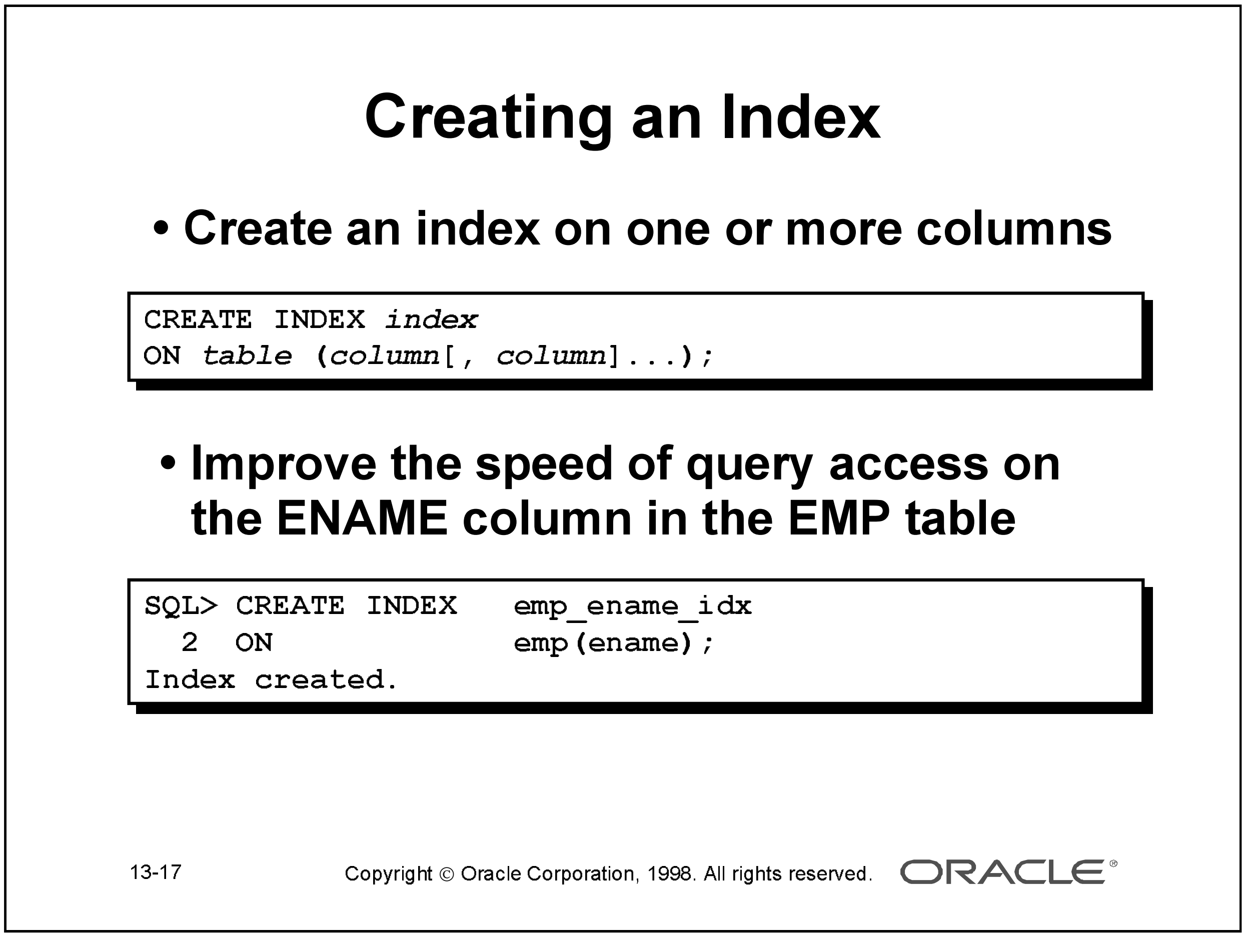
- Automatically
  - A unique index is created automatically when you define a PRIMARY KEY or UNIQUE key constraint in a table definition.
- Manually
  - Users can create nonunique indexes on columns to speed up access time to the rows.

13-16 ORACLE®

How Are Indexes Created?

Two types of indexes can be created. One type is a unique index. The Oracle Server automatically creates this index when you define a column in a table to have a PRIMARY KEY or a UNIQUE key constraint. The name of the index is the name given to the constraint.

The other type of index a user can create is a nonunique index. For example, you can create a FOREIGN KEY column index for a join in a query to improve retrieval speed.



Creating an Index

- **Create an index on one or more columns**

```
CREATE INDEX index
ON table (column[, column]...);
```

- **Improve the speed of query access on the ENAME column in the EMP table**

```
SQL> CREATE INDEX   emp_ename_idx
  2  ON             emp(ename);
Index created.
```

13-17

Creating an Index

Create an index on one or more columns by issuing the CREATE INDEX statement.

In the syntax:

index	is the name of the index.
table	is the name of the table.
column	is the name of the column in the table to be indexed.

For more information, see
Oracle Server SQL Reference, Release 8.0, “CREATE INDEX.”

Guidelines to Creating an Index

- The column is used frequently in the WHERE clause or in a join condition.
- The column contains a wide range of values.
- The column contains a large number of null values.
- Two or more columns are frequently used together in a WHERE clause or a join condition.
- The table is large and most queries are expected to retrieve less than 2–4% of the rows.

13-18 ORACLE®

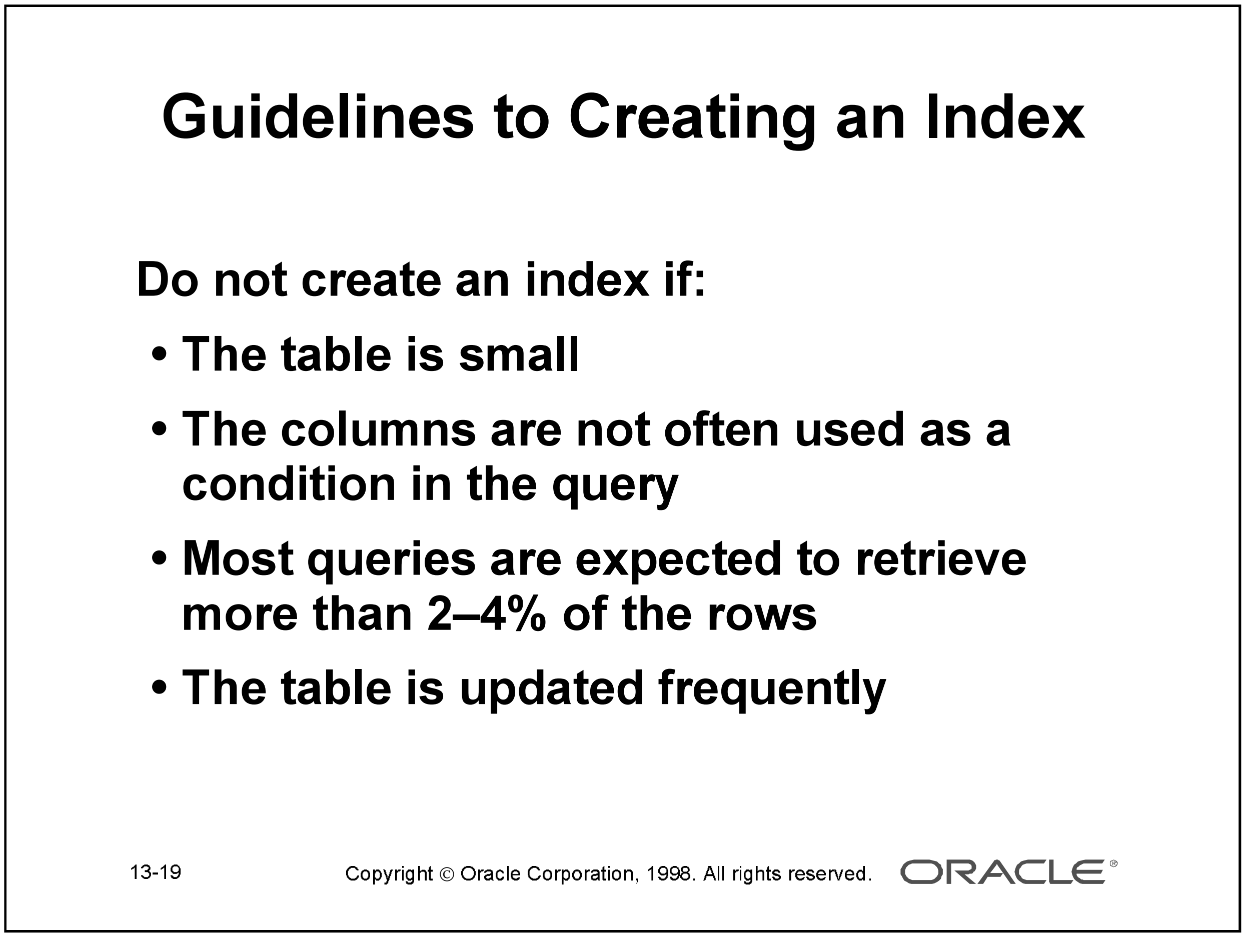
More Is Not Always Better

More indexes on a table does not mean it will speed up queries. Each DML operation that is committed on a table with indexes means that the indexes must be updated. The more indexes you have associated with a table, the more effort the Oracle Server must make to update all the indexes after a DML.

When to Create an Index

- The column is used frequently in the WHERE clause or in a join condition.
- The column contains a wide range of values.
- The column contains a large number of null values.
- Two or more columns are frequently used together in a WHERE clause or join condition.
- The table is large and most queries are expected to retrieve less than 2-4% of the rows.

Remember that if you want to enforce uniqueness, you should define a unique constraint in the table definition. Then, a unique index is automatically created.



Guidelines to Creating an Index

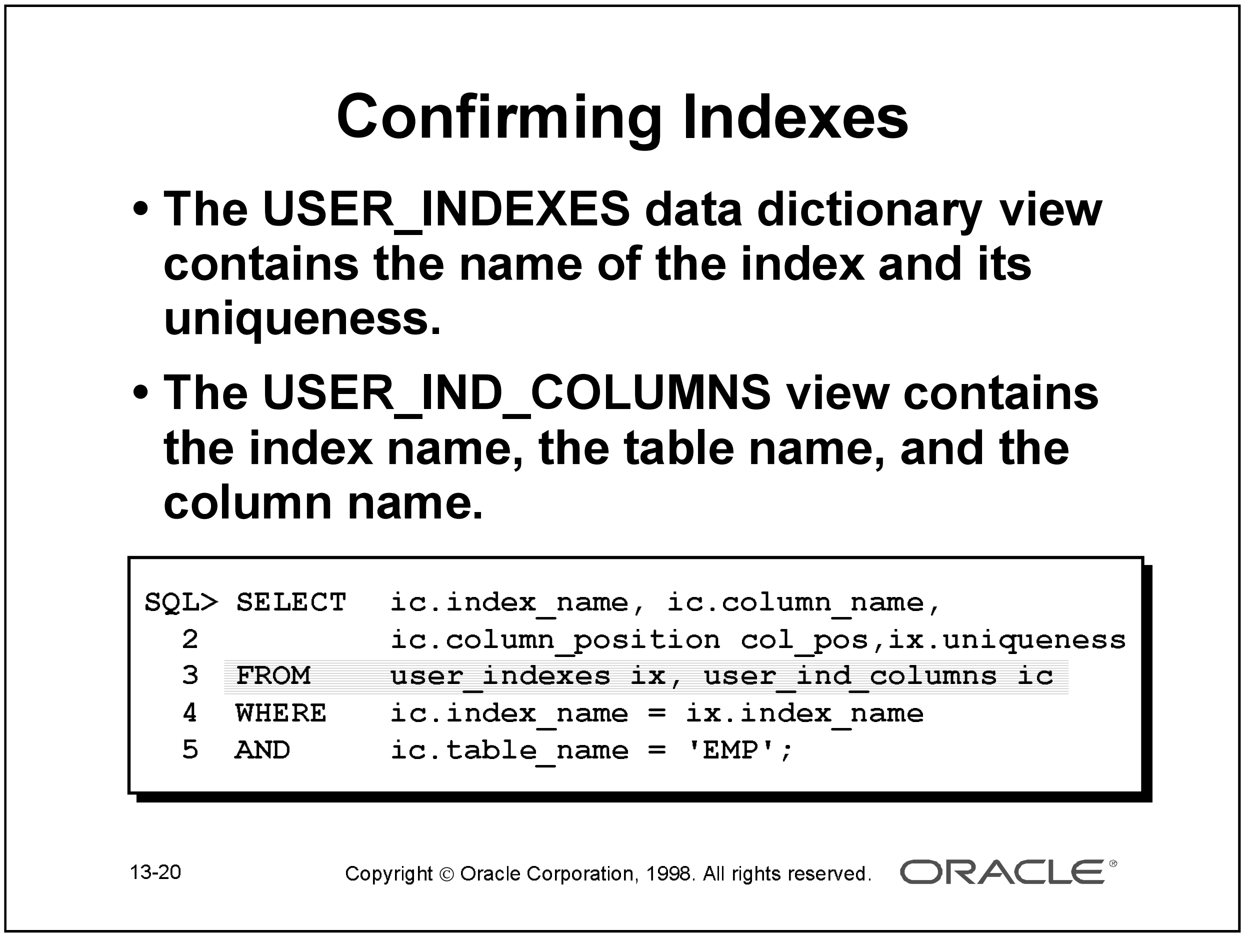
Do not create an index if:

- The table is small
- The columns are not often used as a condition in the query
- Most queries are expected to retrieve more than 2–4% of the rows
- The table is updated frequently

13-19

When to Not Create an Index

- The table is small.
- The columns are not often used as a condition in the query.
- Most queries are expected to retrieve more than 2–4% of the rows.
- The table is updated frequently. If you have one or more indexes on a table, the DML statements that access the table take relatively more time due to maintenance of indexes.



Confirming Indexes

- The USER_INDEXES data dictionary view contains the name of the index and its uniqueness.
- The USER_IND_COLUMNS view contains the index name, the table name, and the column name.

```
SQL> SELECT   ic.index_name, ic.column_name,
              ic.column_position col_pos,ix.uniqueness
     FROM     user_indexes ix, user_ind_columns ic
     WHERE    ic.index_name = ix.index_name
     AND      ic.table_name = 'EMP';
```

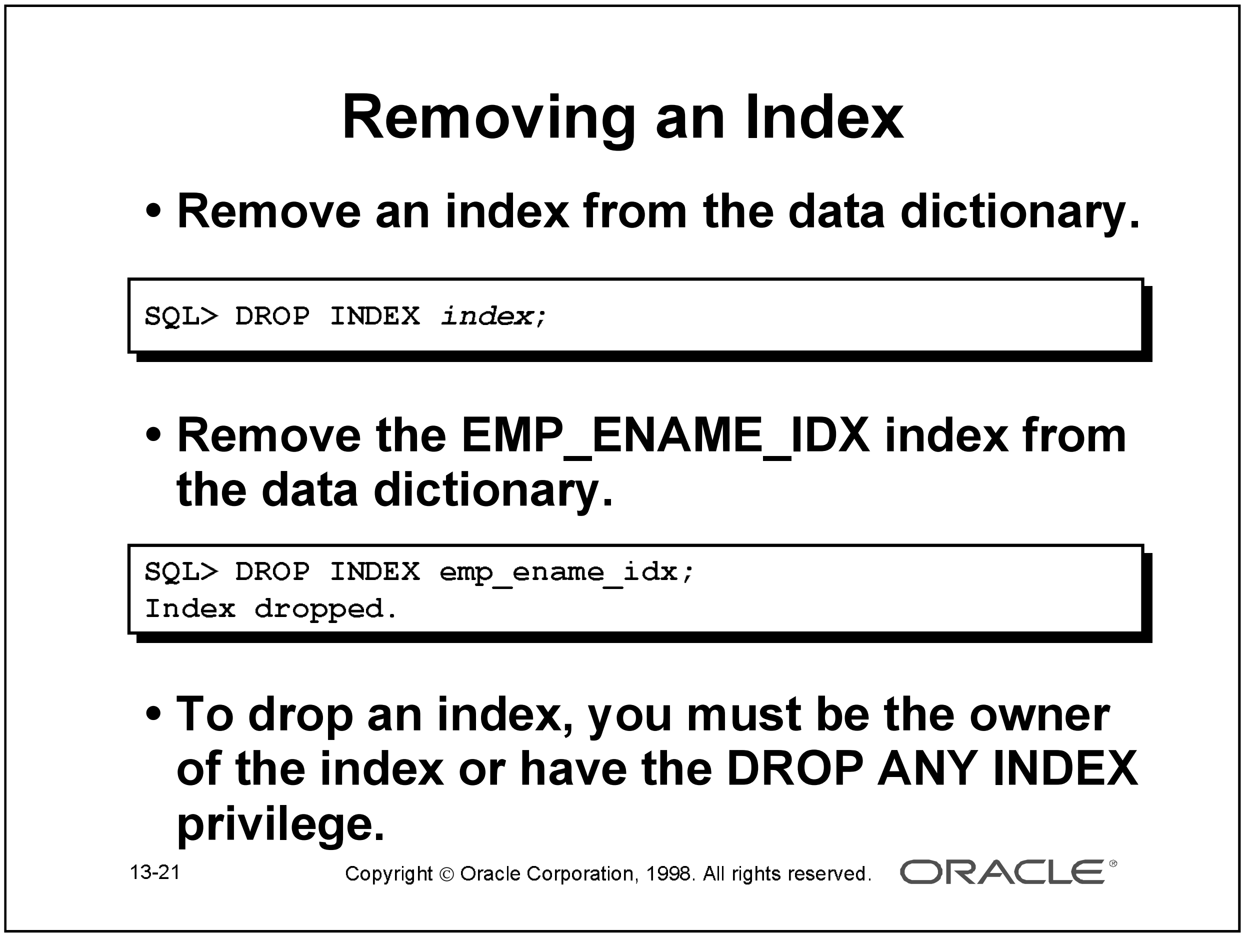
Confirming Indexes

Confirm the existence of indexes from the USER_INDEXES data dictionary view. You can also check the columns involved in an index by querying the USER_IND_COLUMNS view.

The example above displays all the previously created indexes, affected column names, and uniqueness on the EMP table.

```
INDEX_NAME         COLUMN_NAME     COL_POS UNIQUENES
-----------------  --------------- ------- ----------
EMP_EMPNO_PK       EMPNO                 1 UNIQUE
EMP_ENAME_IDX      ENAME                 1 NONUNIQUE
```

Note: The output has been formatted.



Removing an Index

- Remove an index from the data dictionary.

```
SQL> DROP INDEX index;
```

- Remove the EMP_ENAME_IDX index from the data dictionary.

```
SQL> DROP INDEX emp_ename_idx;
Index dropped.
```

- To drop an index, you must be the owner of the index or have the DROP ANY INDEX privilege.

13-21

Removing an Index

You cannot modify indexes. To change an index, you must drop it and then re-create it. Remove an index definition from the data dictionary by issuing the DROP INDEX statement. To drop an index, you must be the owner of the index or have the DROP ANY INDEX privilege.

In the syntax:

index	is the name of the index.

Synonyms

Simplify access to objects by creating a synonym (another name for an object).

- **Refer to a table owned by another user.**
- **Shorten lengthy object names.**

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```

13-22

Creating a Synonym for an Object

To refer to a table owned by another user, you need to prefix the table name with the name of the user who created it followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

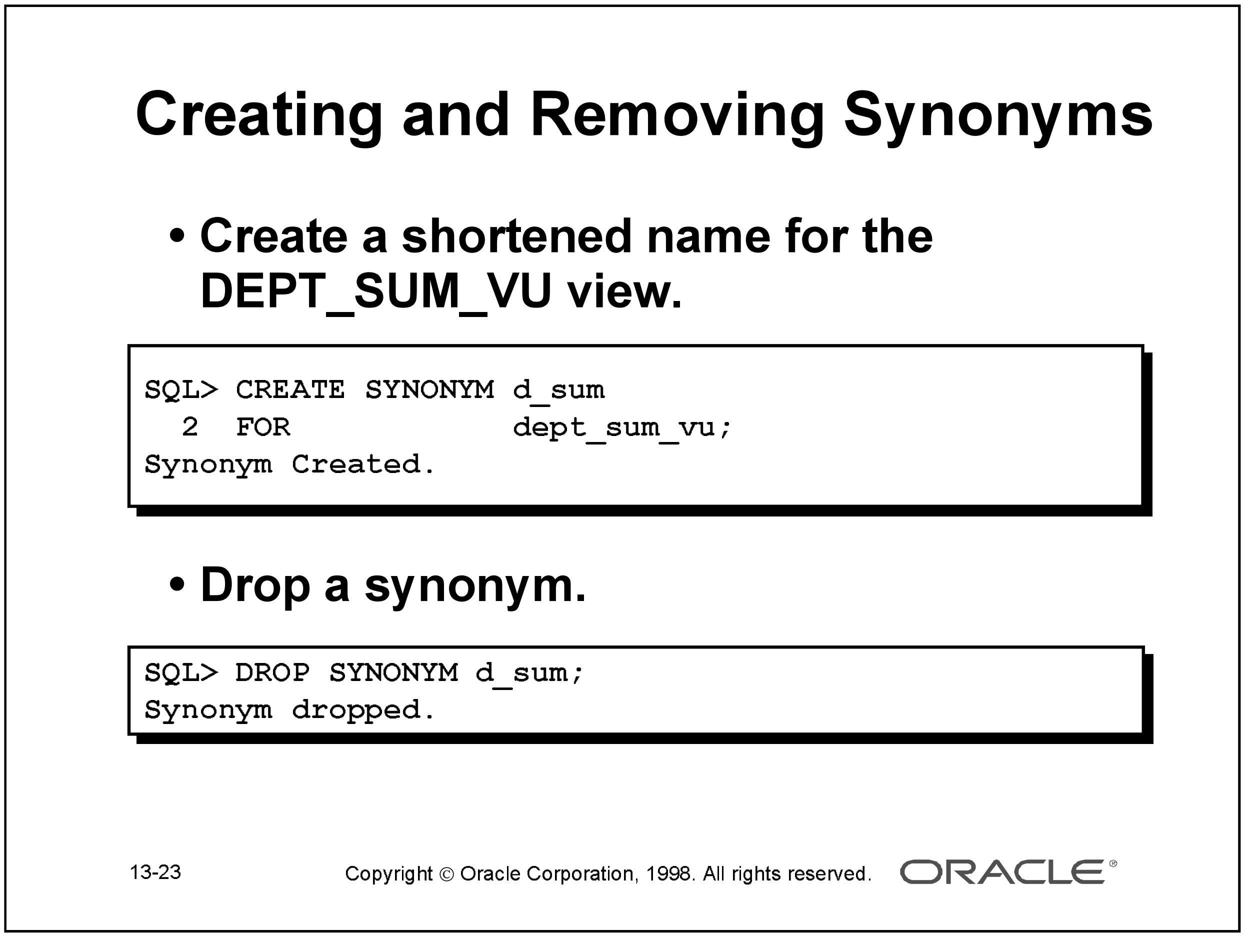
In the syntax:

PUBLIC	creates a synonym accessible to all users.
synonym	is the name of the synonym to be created.
object	identifies the object for which the synonym is created.

Guidelines

- The object cannot be contained in a package.
- A private synonym name must be distinct from all other objects owned by the same user.

For more information, see
Oracle Server SQL Reference, Release 8.0, "CREATE SYNONYM."



Creating and Removing Synonyms

- **Create a shortened name for the DEPT_SUM_VU view.**

```
SQL> CREATE SYNONYM d_sum
  2  FOR           dept_sum_vu;
Synonym Created.
```

- **Drop a synonym.**

```
SQL> DROP SYNONYM d_sum;
Synonym dropped.
```

13-23 ORACLE®

Creating a Synonym for an Object (continued)

The example above creates a synonym for the DEPT_SUM_VU view for quicker reference.

The DBA can create a public synonym accessible to all users. The example below creates a public synonym named DEPT for Alice's DEPT table:

```
SQL> CREATE PUBLIC SYNONYM  dept
  2  FOR                    alice.dept;
Synonym created.
```

Removing a Synonym

To drop a synonym, use the DROP SYNONYM statement. Only the DBA can drop a public synonym.

```
SQL> DROP SYNONYM  dept;
Synonym dropped.
```

For more information, see
Oracle Server SQL Reference, Release 8.0, "DROP SYNONYM."

Summary

- Automatically generate sequence numbers by using a sequence generator.
- View sequence information in the USER_SEQUENCES data dictionary table.
- Create indexes to improve query retrieval speed.
- View index information in the USER_INDEXES dictionary table.
- Use synonyms to provide alternative names for objects.

Sequences

The sequence generator can be used to automatically generate sequence numbers for rows in tables. This can be time saving and can reduce the amount of application code needed.

A sequence is a database object that can be shared with other users. Information about the sequence can be found in the USER_SEQUENCES table of the data dictionary.

To use a sequence, reference it with either the NEXTVAL or the CURRVAL pseudocolumns.

- Retrieve the next number in the sequence by referencing *sequence*.NEXTVAL.
- Return the current available number by referencing *sequence*.CURRVAL.

Indexes

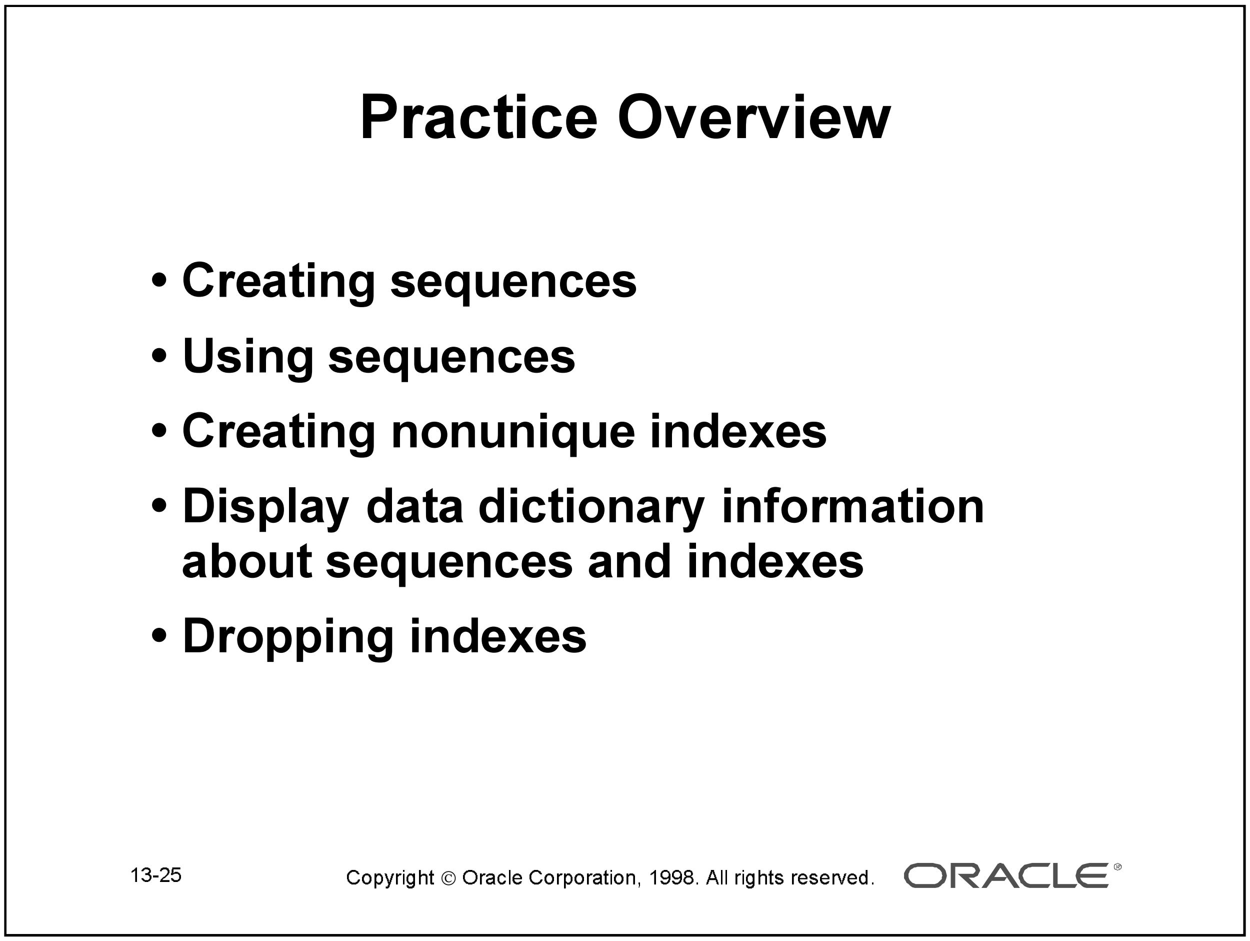
Indexes are used to improve the query retrieval speed.

Users can view the definitions of the indexes in the USER_INDEXES data dictionary view.

An index can be dropped by the creator or a user with the DROP ANY INDEX privilege by using the DROP INDEX statement.

Synonyms

DBAs can create public synonyms, and users can create private synonyms for convenience by using the CREATE SYNONYM statement. Synonyms permit short names or alternative names for objects. Remove synonyms by using the DROP SYNONYM statement.



Practice Overview

- Creating sequences
- Using sequences
- Creating nonunique indexes
- Display data dictionary information about sequences and indexes
- Dropping indexes

13-25

Practice Overview

In this practice, you will create a sequence to be used when populating your DEPARTMENT table. You will also create implicit and explicit indexes.

Practice 13

1. Create a sequence to be used with the DEPARTMENT table's primary key column. The sequence should start at 60 and have a maximum value of 200. Have your sequence increment by ten numbers. Name the sequence DEPT_ID_SEQ.
2. Write a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script *p13q2.sql*. Execute your script.

```
SEQUENCE_NAME MAX_VALUE INCREMENT_BY  LAST_NUMBER
------------- --------- ------------  -----------
CUSTID        1.000E+27            1          109
DEPT_ID_SEQ         200            1           60
ORDID         1.000E+27            1          622
PRODID        1.000E+27            1       200381
```

3. Write an interactive script to insert a row into the DEPARTMENT table. Name your script *p13q3.sql*. Be sure to use the sequence that you created for the ID column. Create a customized prompt to enter the department name. Execute your script. Add two departments named Education and Administration. Confirm your additions.
4. Create a non-unique index on the FOREIGN KEY column in the EMPLOYEE table.
5. Display the indexes and uniqueness that exist in the data dictionary for the EMPLOYEE table. Save the statement into a script named *p13q5.sql*.

```
INDEX_NAME           TABLE_NAME   UNIQUENES
-------------------- ------------ ---------
EMPLOYEE_DEPT_ID_IDX EMPLOYEE     NONUNIQUE
EMPLOYEE_ID_PK       EMPLOYEE     UNIQUE
```

14

Controlling User Access

Objectives

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

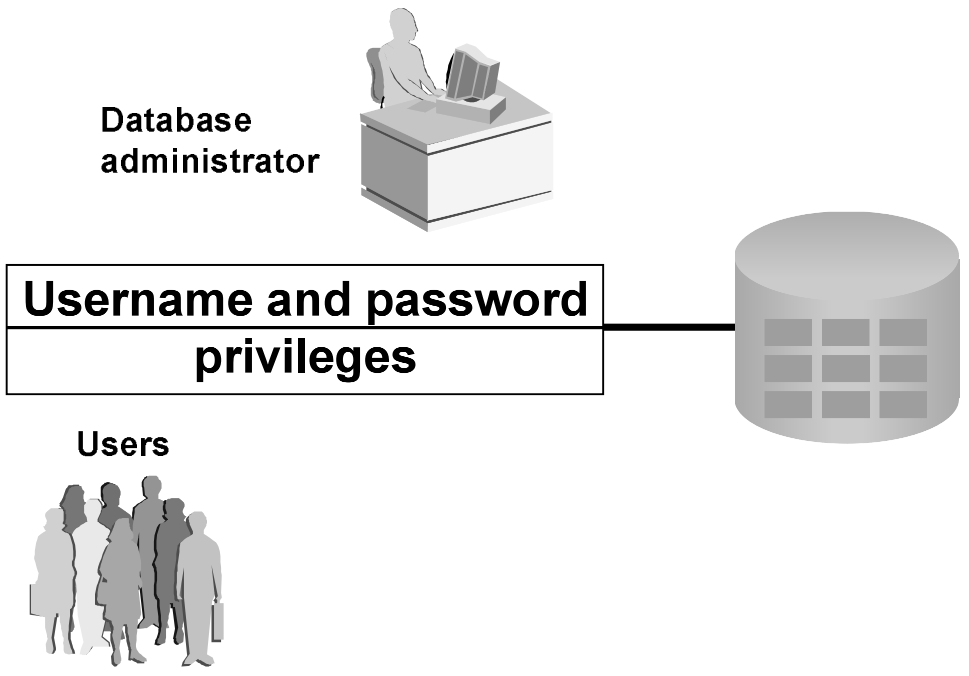
- Create users
- Create roles to ease setup and maintenance of the security model
- GRANT and REVOKE object privileges

14-2

Lesson Aim

In this lesson, you will learn how to control database access to specific objects and add new users with different levels of access privileges.

Controlling User Access

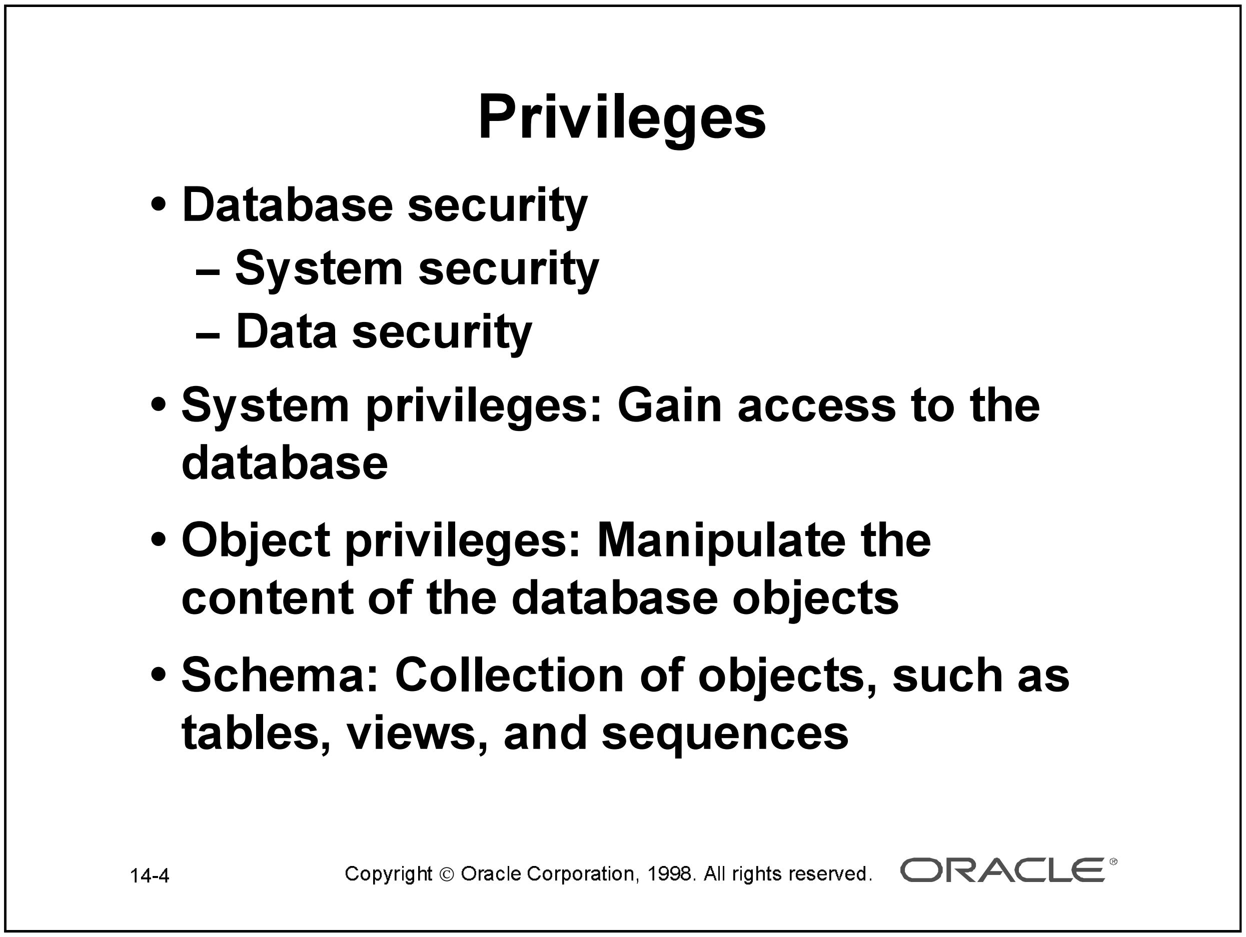


Controlling User Access

In a multiple-user environment, you want to maintain security of the database access and use. Oracle Server database security allows you to do the following:

- Control database access
- Give access to specific objects in the database
- Confirm given and received *privileges* with the Oracle data dictionary
- Create synonyms for database objects

Database security can be classified into two categories: system security and data security. System security covers access and use of the database at the system level, such as username and password, disk space allocated to users, and system operations allowed by the user. Database security covers access and use of the database objects and the actions that those users can have on the objects.



Privileges

- **Database security**
  - **System security**
  - **Data security**
- **System privileges: Gain access to the database**
- **Object privileges: Manipulate the content of the database objects**
- **Schema: Collection of objects, such as tables, views, and sequences**

14-4

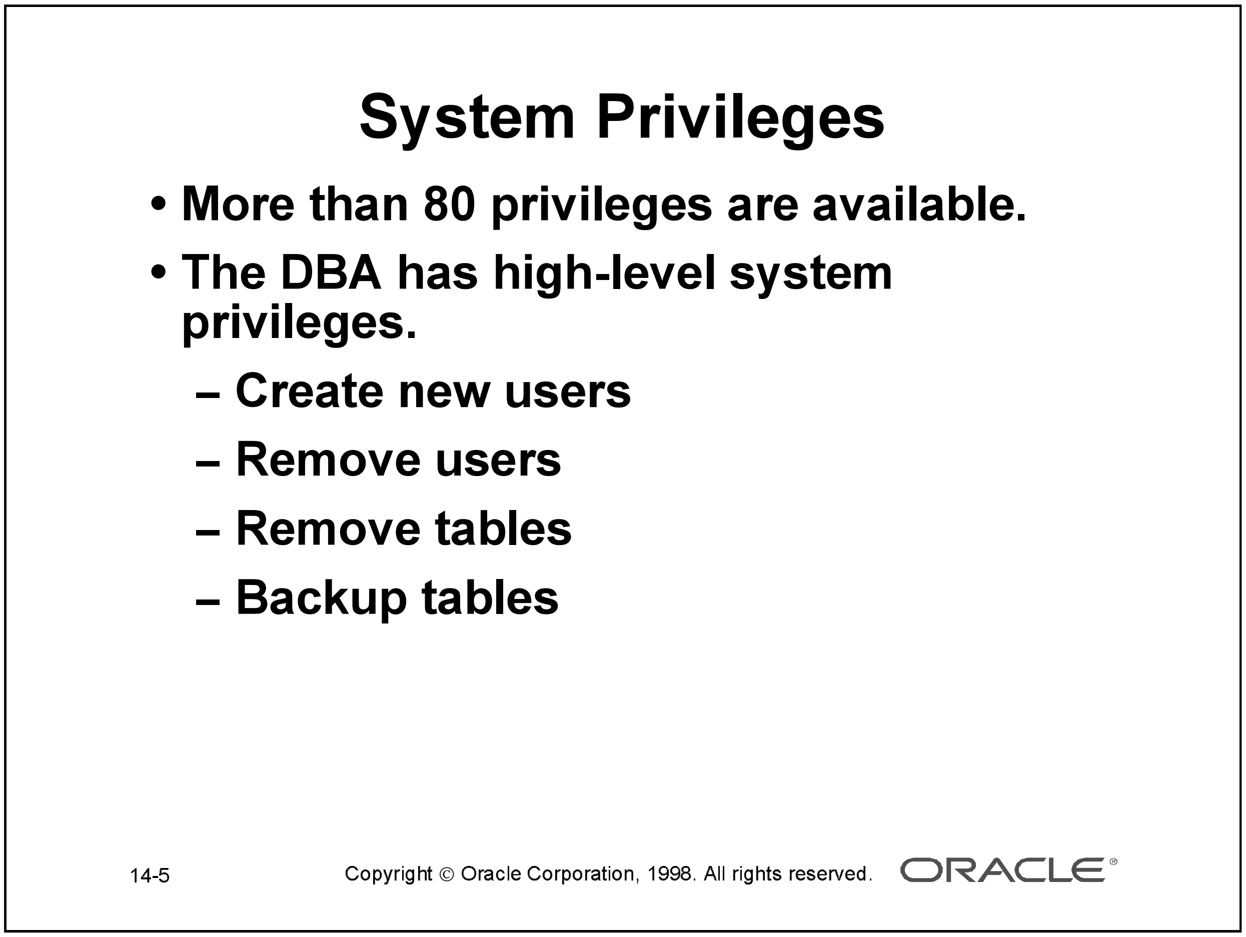
Privileges

Privileges are the right to execute particular SQL statements. The database administrator is a high-level user with the ability to grant users access to the database and its objects. The users require *system privileges* to gain access to the database and *object privileges* to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to *roles*, which are named groups of related privileges.

Schema

A ***schema*** is a collection of objects, such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.

For more information,
see *Oracle Server Application Developer's Guide, Release 8.0*, "Establishing a Security Policy" section, and *Oracle Server Concepts Manual*, *Release 8.0*, "Database Security" topic.



System Privileges

- More than 80 privileges are available.
- The DBA has high-level system privileges.
  - Create new users
  - Remove users
  - Remove tables
  - Backup tables

14-5

System Privileges

More than 80 system privileges are available for users and roles. System privileges are typically provided by the database administrator.

Typical DBA Privileges

System Privilege	Operations Authorized
CREATE USER	Allows grantee to create other Oracle users (a privilege required for a DBA role)
DROP USER	Drops another user
DROP ANY TABLE	Drops a table in any schema
BACKUP ANY TABLE	Backs up any table in any schema with the export utility

Creating Users

The DBA creates users by using the CREATE USER statement.

```
CREATE USER     user
IDENTIFIED BY  password;
```

```
SQL> CREATE  USER  scott
  2  IDENTIFIED BY tiger;
User created.
```

Creating a User

The DBA creates the user by executing the CREATE USER statement. The user does not have any privileges at this point. The DBA can then grant a number of privileges to that user. These privileges determine what the user can do at the database level.

The slide gives the abridged syntax for creating a user.

In the syntax:

user	is the name of the user to be created.
password	specifies that the user must log in with this password.

For more information,
see *Oracle Server SQL Reference, Release 8.0*, “GRANT” (System Privileges and Roles) and “CREATE USER.”

User System Privileges

- Once a user is created, the DBA can grant specific system privileges to a user.

```
GRANT privilege [, privilege...]
TO user [, user...];
```

- An application developer may have the following system privileges:
  - CREATE SESSION
  - CREATE TABLE
  - CREATE SEQUENCE
  - CREATE VIEW
  - CREATE PROCEDURE

Typical User Privileges

Now that the DBA has created a user, the DBA can assign privileges to that user.

System Privilege	Operations Authorized
CREATE SESSION	Connect to the database
CREATE TABLE	Create tables in the user's schema
CREATE SEQUENCE	Create a sequence in the user's schema
CREATE VIEW	Create a view in the user's schema
CREATE PROCEDURE	Create a stored procedure, function, or package in the user's schema

In the syntax:

privilege	is the system privilege to be granted.
user	is the name of the user.

Granting System Privileges

The DBA can grant a user specific system privileges.

```
SQL> GRANT  create table, create sequence, create view
  2  TO     scott;
Grant succeeded.
```

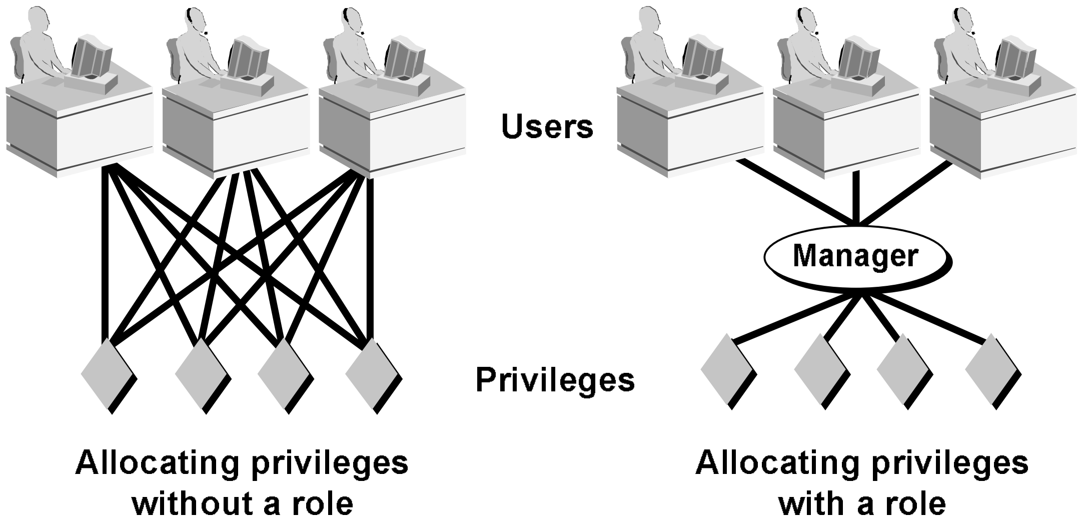
14-8 ORACLE®

Granting System Privileges

The DBA uses the GRANT statement to allocate system privileges to the user. Once the user has been granted the privileges, the user can immediately use those privileges.

In the above example, user Scott has been assigned the privileges to create tables, sequences, and views.

What Is a Role?


Users
Manager
Privileges
Allocating privileges without a role
Allocating privileges with a role

14-9 ORACLE®

What Is a Role?

A role is a named group of related privileges that can be granted to the user. This method makes granting and revoking privileges easier to perform and maintain.

A user can have access to several roles, and several users can be assigned the same role. Roles typically are created for a database application.

Creating and Assigning a Role

First, the DBA must create the role. Then the DBA can assign privileges to the role and users to the role.

Syntax

```
CREATE   ROLE  role;
```

where:	*role*	is the name of the role to be created.

Now that the role is created, the DBA can use the GRANT statement to assign users to the role as well as assign privileges to the role.

Creating and Granting Privileges to a Role

```
SQL> CREATE ROLE manager;
Role created.
```

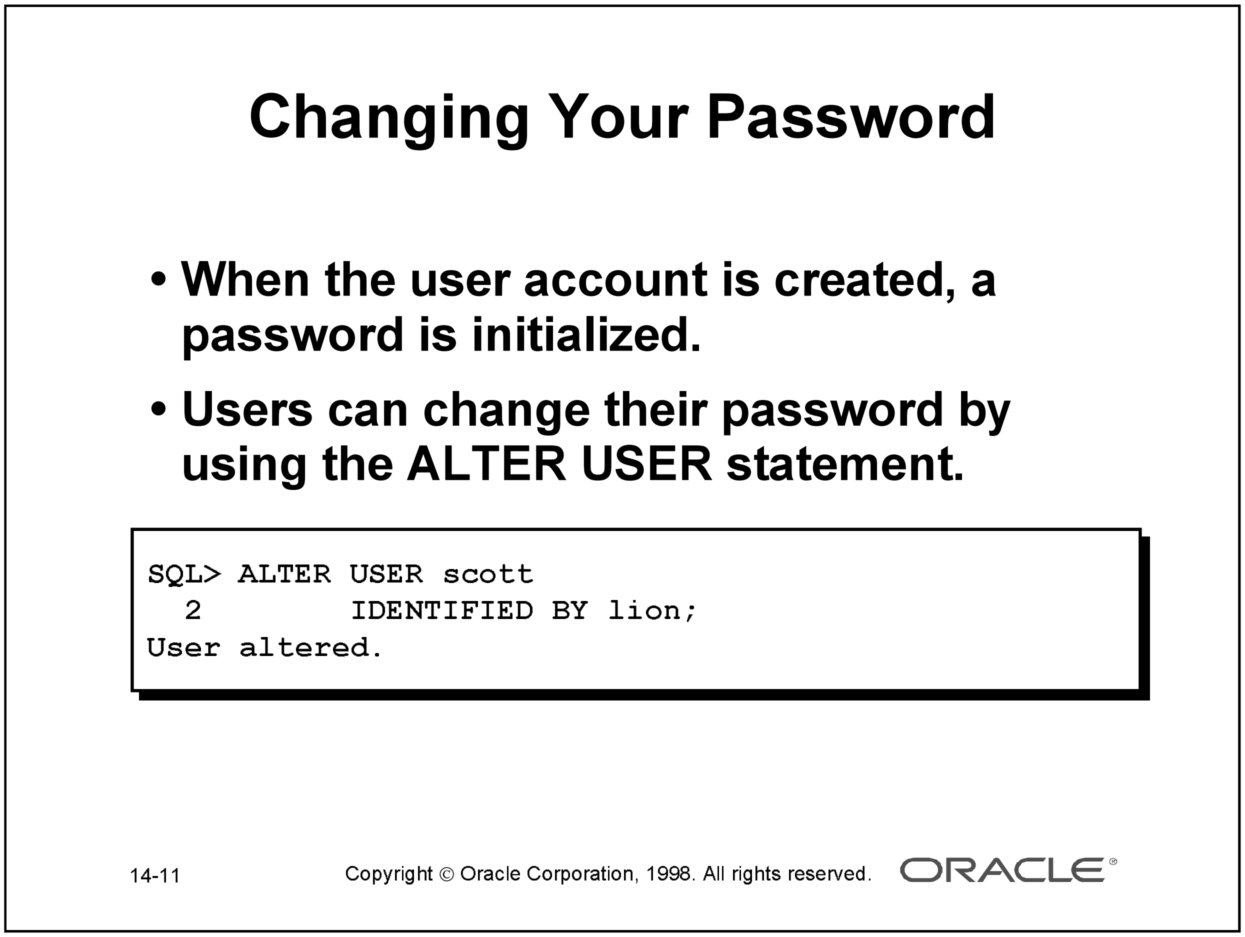
```
SQL> GRANT create table, create view
  2        to manager;
Grant succeeded.
```

```
SQL> GRANT manager to BLAKE, CLARK;
Grant succeeded.
```

14-10 ORACLE®

Creating a Role

The example above creates a role manager and then allows the managers to create tables and views. It then grants Blake and Clark the role of managers. Now Blake and Clark can create tables and views.



Changing Your Password

- When the user account is created, a password is initialized.
- Users can change their password by using the ALTER USER statement.

```
SQL> ALTER USER scott
  2        IDENTIFIED BY lion;
User altered.
```

14-11

Changing Your Password

Every user has a password that is initialized by the DBA when the user is created. You can change your password by using the ALTER USER statement.

Syntax

```
ALTER USER user IDENTIFIED BY password;
```

where:	*user*	is the name of the user.
	password	specifies the new password.

Although this statement can be used to change your password, there are many other options. You must have the ALTER USER privilege to change any other option.

For more information,
see *Oracle Server SQL Reference, Release 8.0*, “ALTER USER.”

Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER	√		√	
DELETE	√	√		
EXECUTE				√
INDEX	√			
INSERT	√	√		
REFERENCES	√			
SELECT	√	√	√	
UPDATE	√	√		

14-12

Object Privileges

An *object privilege* is a privilege or right to perform a particular action on a specific table, view, sequence, or procedure. Each object has a particular set of grantable privileges. The table above lists the privileges for various objects. Note that the only privileges that apply to a sequence are SELECT and ALTER. UPDATE, REFERENCES, and INSERT can be restricted by specifying a subset of updatable columns. A SELECT can be restricted by creating a view with a subset of columns and granting SELECT privilege on the view. A grant on a synonym is converted to a grant on the base table referenced by the synonym.

Object Privileges

- **Object privileges vary from object to object.**
- **An owner has all the privileges on the object.**
- **An owner can give specific privileges on that owner's object.**

```
GRANT          object_priv [(columns)]
ON             object
TO             {user|role|PUBLIC}
[WITH GRANT OPTION];
```

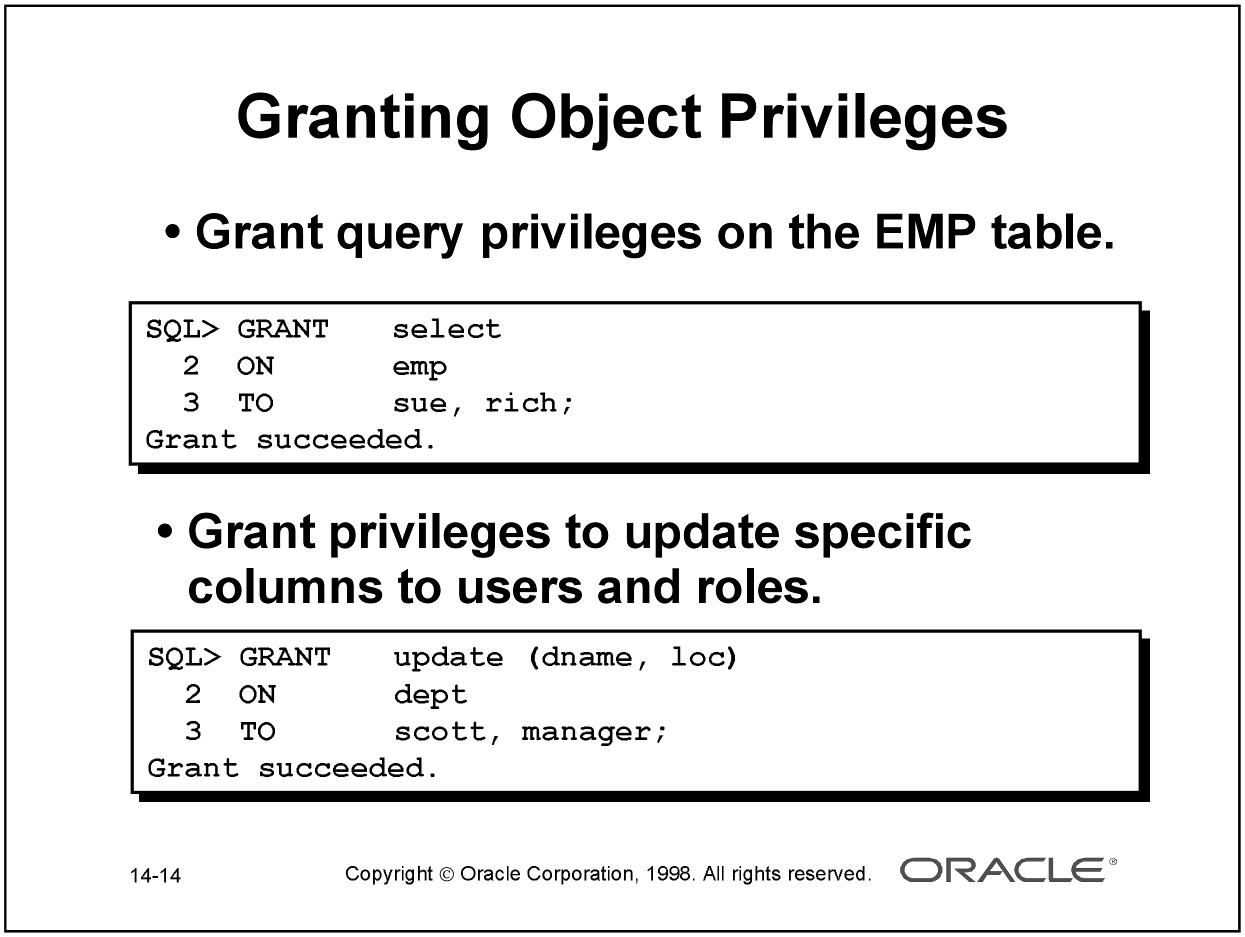
14-13

Granting Object Privileges

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in his or her schema. A user can grant any object privilege on any schema object he or she owns to any other user or role. If the grant includes the GRANT OPTION, the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

In the syntax:

object_priv	is an object privilege to be granted.
ALL	all object privileges.
columns	specifies the column from a table or view on which privileges are granted.
ON *object*	is the object on which the privileges are granted.
TO	identifies to whom the privilege is granted.
PUBLIC	grants object privileges to all users.
WITH GRANT OPTION	allows the grantee to grant the object privileges to other users and roles.



Granting Object Privileges

- Grant query privileges on the EMP table.

```
SQL> GRANT    select
  2  ON       emp
  3  TO       sue, rich;
Grant succeeded.
```

- Grant privileges to update specific columns to users and roles.

```
SQL> GRANT    update (dname, loc)
  2  ON       dept
  3  TO       scott, manager;
Grant succeeded.
```

Guidelines

- To grant privileges on an object, the object must be in your own schema or you must have been granted the object privileges WITH GRANT OPTION.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

The first example above grants users Sue and Rich the privilege to query your EMP table. The second example grants UPDATE privileges on specific columns in the DEPT table to Scott and to the manager role.

Note: DBAs generally allocate system privileges; any user who owns an object can grant object privileges.

Using WITH GRANT OPTION and PUBLIC Keywords

- **Give a user authority to pass along the privileges.**

```
SQL> GRANT    select, insert
  2  ON       dept
  3  TO       scott
  4  WITH GRANT OPTION;
Grant succeeded.
```

- **Allow all users on the system to query data from Alice’s DEPT table.**

```
SQL> GRANT    select
  2  ON       alice.dept
  3  TO       PUBLIC;
Grant succeeded.
```

WITH GRANT OPTION Keyword

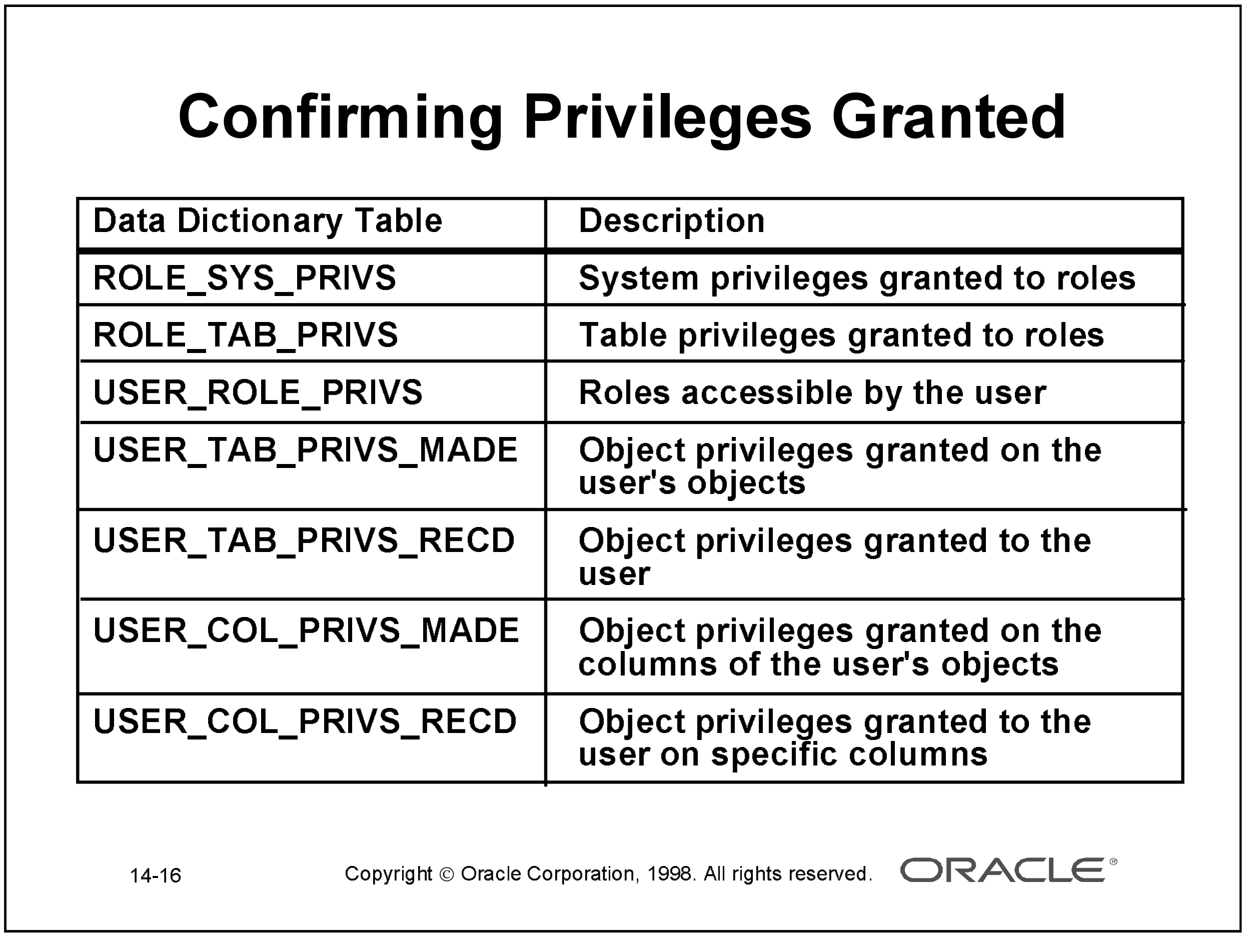
A privilege that is granted WITH GRANT OPTION can be passed on to other users and roles by the grantee. Object privileges granted WITH GRANT OPTION are revoked when the grantor’s privilege is revoked.

The above example allows user Scott to access your DEPT table with the privileges to query the table and add rows to the table. Allow Scott to give others these privileges.

PUBLIC Keyword

An owner of a table can grant access to all users by using the PUBLIC keyword.

The above example allows all users on the system to query data from Alice’s DEPT table.



Confirming Privileges Granted

Data Dictionary Table	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_RECD	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns

14-16 ORACLE®

Confirming Privileges Granted

If you attempt to perform an unauthorized operation—for example, deleting a row from a table for which you do not have the DELETE privilege—the Oracle Server will not permit the operation to take place.

If you receive the Oracle Server error message “table or view does not exist,” you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege

You can access the data dictionary to view the privileges that you have. The table on the slide describes various data dictionary tables.

How to Revoke Object Privileges

- You use the REVOKE statement to revoke privileges granted to other users.
- Privileges granted to others through the WITH GRANT OPTION will also be revoked.

```
REVOKE {privilege [, privilege...]|ALL}
ON     object
FROM   {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```

14-17

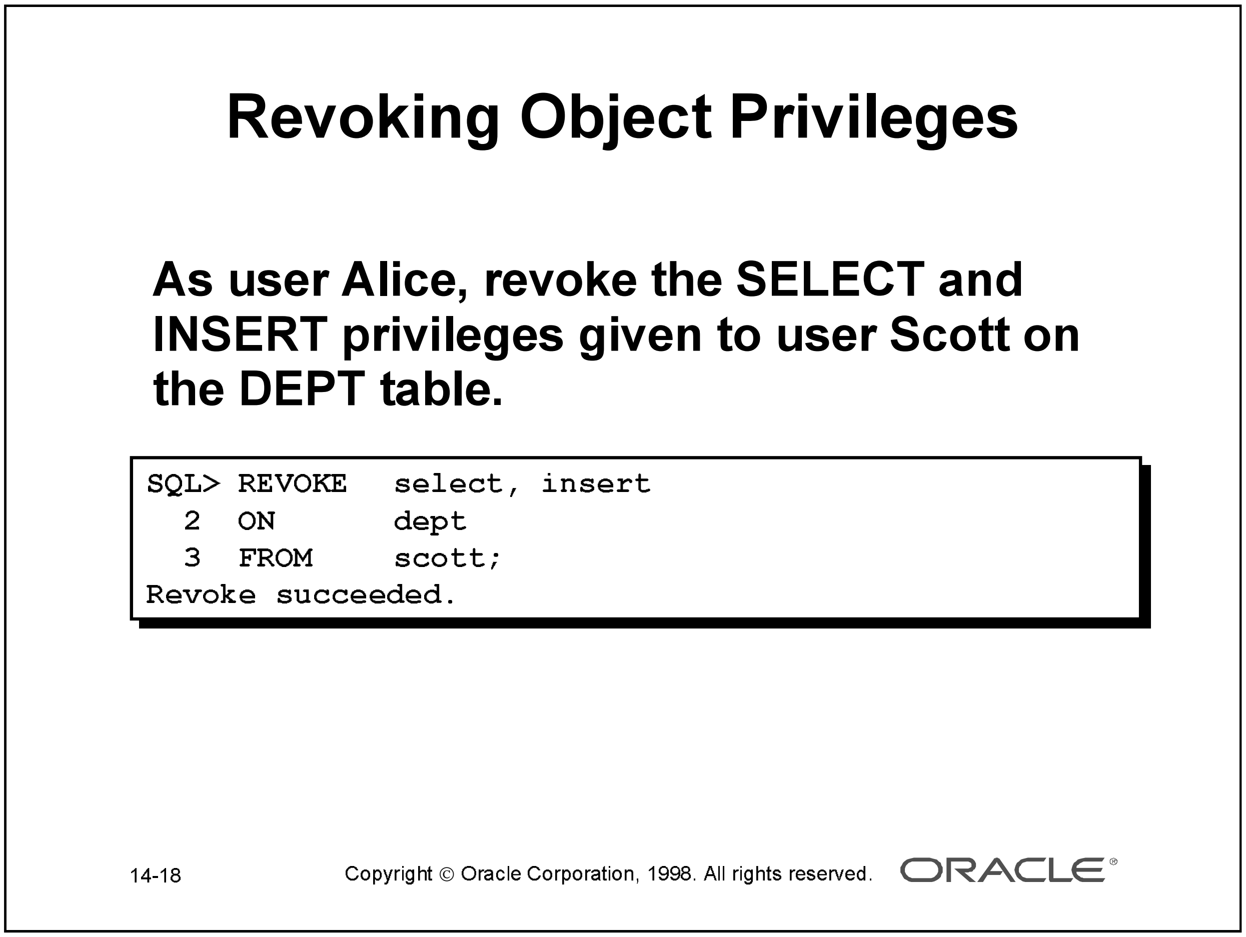
Revoking Object Privileges

Remove privileges granted to other users by using the REVOKE statement. When you use the REVOKE statement, the privileges that you specify are revoked from the users that you name and from any other users to whom those privileges may have been granted.

In the syntax:

CASCADE CONSTRAINTS	is required to remove any referential integrity constraints made to the object by means of the REFERENCES privilege.

For more information,
see *Oracle Server SQL Reference, Release 8.0*, “REVOKE.”



Revoking Object Privileges

As user Alice, revoke the SELECT and INSERT privileges given to user Scott on the DEPT table.

```
SQL> REVOKE   select, insert
  2  ON       dept
  3  FROM     scott;
Revoke succeeded.
```

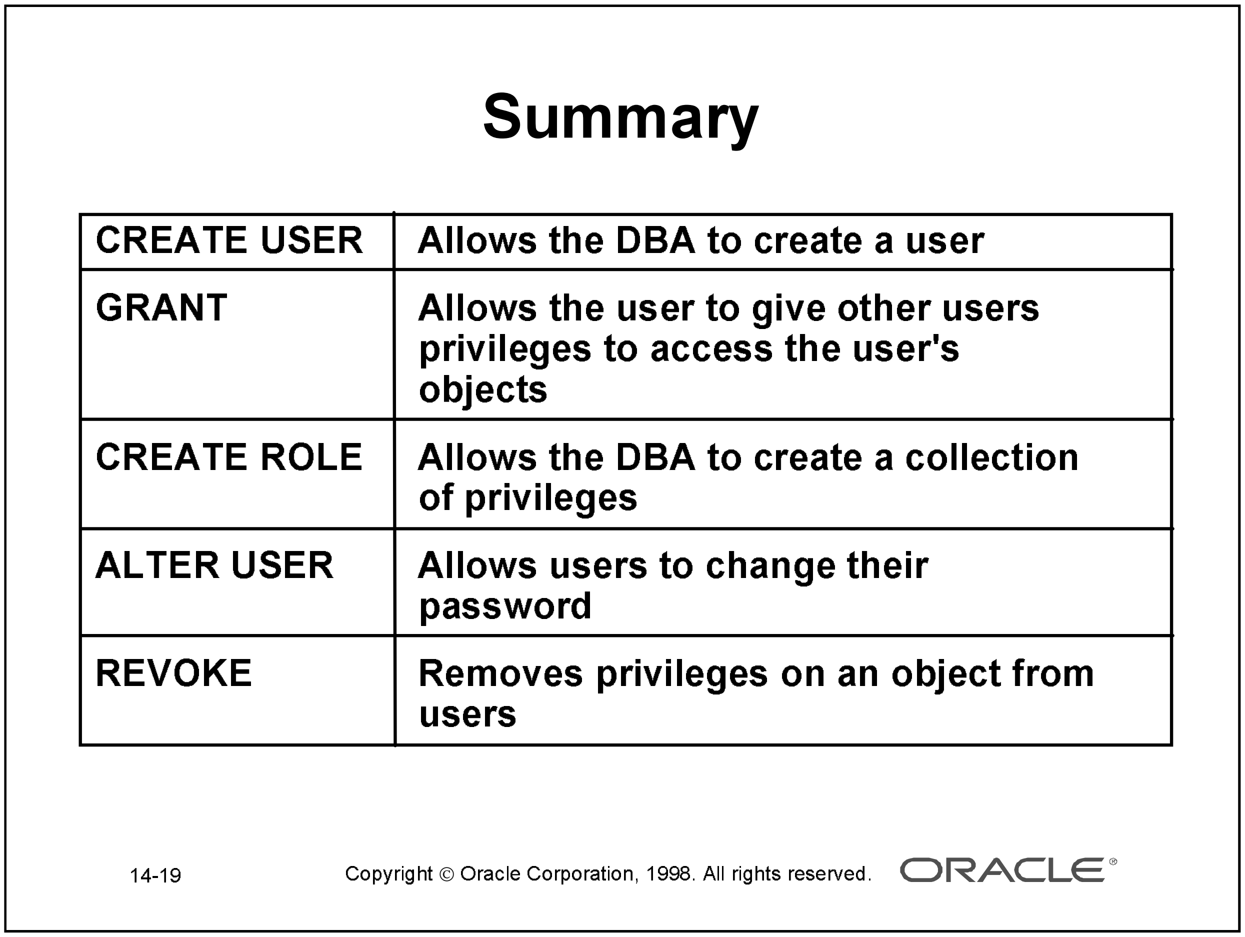
14-18

Revoking Object Privileges (continued)

The example above revokes SELECT and INSERT privileges given to user Scott on the DEPT table.

Note: If a user is granted a privilege WITH GRANT OPTION, that user can also grant the privilege WITH GRANT OPTION, so that a long chain of grantees is possible, but no circular grants are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, the REVOKE cascades to all privileges granted.

For example, if user A grants SELECT privilege on a table to user B including the WITH GRANT OPTION, user B can grant to user C the SELECT privilege WITH GRANT OPTION, and user C can then grant to user D the SELECT privilege. If user A the revokes the privilege from user B, then the privileges granted to users C and D are also revoked.



Summary

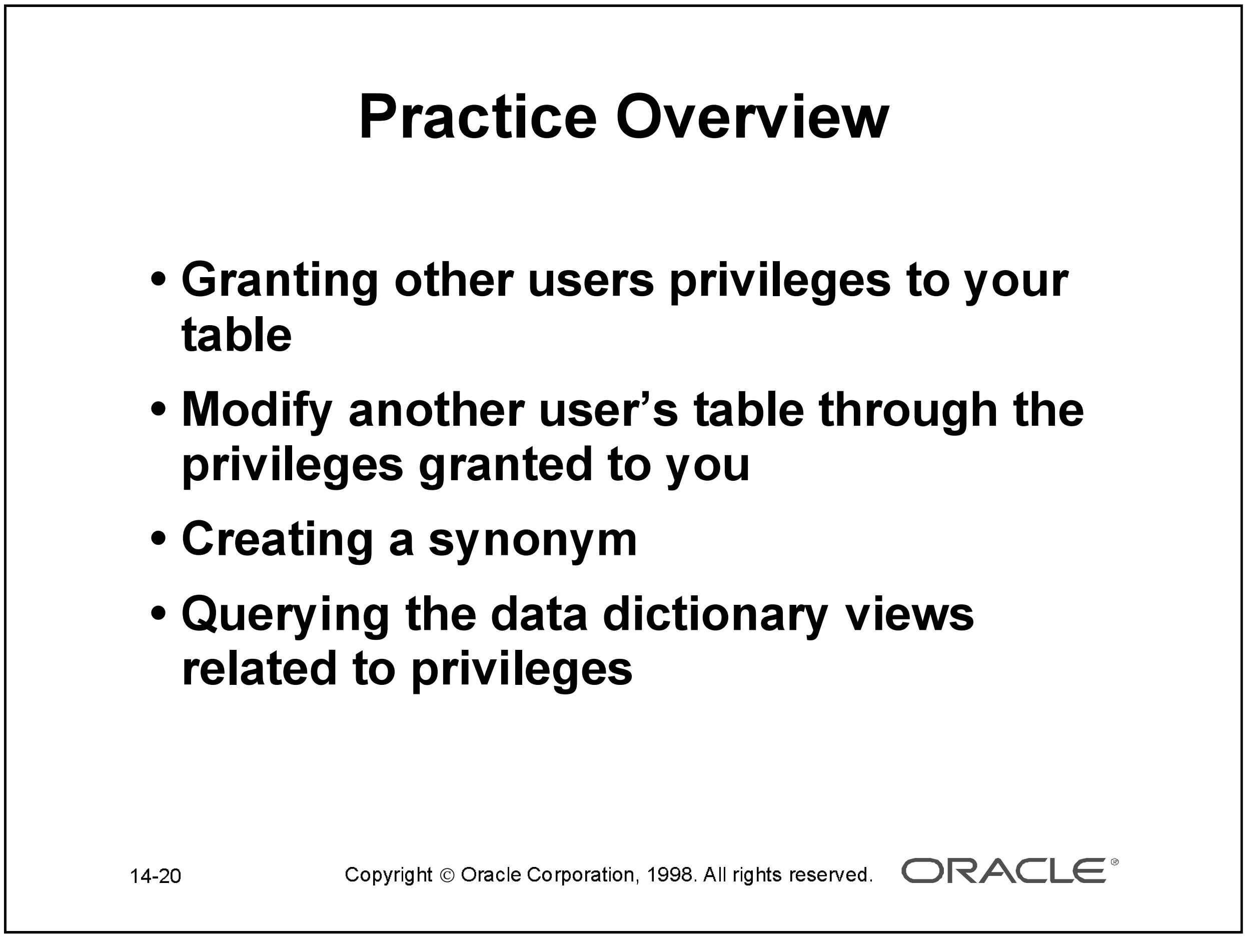
CREATE USER	Allows the DBA to create a user
GRANT	Allows the user to give other users privileges to access the user's objects
CREATE ROLE	Allows the DBA to create a collection of privileges
ALTER USER	Allows users to change their password
REVOKE	Removes privileges on an object from users

14-19 ORACLE®

Summary

DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- Once the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the GRANT statement.
- A DBA can create roles by using the CREATE ROLE statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their password by using the ALTER USER statement.
- You can remove privileges from users by using the REVOKE statement.
- Data dictionary views allow users to view the privileges granted to them and those that are granted on their objects.



Practice Overview

- Granting other users privileges to your table
- Modify another user’s table through the privileges granted to you
- Creating a synonym
- Querying the data dictionary views related to privileges

14-20

Practice Overview

Team up with other students for this exercise of controlling access to database objects.

Practice 14

1. What privilege should a user be given to log in to the Oracle Server? Is this privilege a system or object privilege?

   ____

2. What privilege should a user be given to create tables?

   ____

3. If you create a table, who can pass along privileges to other users on your table?

   ____

4. You are the DBA. You are creating many users who require the same system privileges. What would you use to make your job easier?

   ____

5. What command do you use to change your password?

   ____

6. Grant another user access to your DEPT table. Have the user grant you query access to his or her DEPT table.
7. Query all the rows in your DEPT table.

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

8. Add a new row to your DEPT table. Team 1 should add Education as department number 50. Team 2 should add Administration as department number 50. Make the changes permanent.
9. Create a synonym for the other team's DEPT table.

Practice 14 (continued)

10. Query all the rows in the other team's DEPT table by using your synonym.

```
Team 1 SELECT statement results.
DEPTNO DNAME          LOC
------ -------------- ---------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
    50 ADMINISTRATION
Team 2 SELECT statement results.
DEPTNO DNAME          LOC
------ -------------- ---------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
    50 EDUCATION
```

11. Query the USER_TABLES data dictionary to see information about the tables that you own.

```
TABLE_NAME
----------------
BONUS
CUSTOMER
DEPARTMENT
DEPT
DUMMY
EMP
EMPLOYEE
ITEM
MY_EMPLOYEE
ORD
PRICE
PRODUCT
SALGRADE
13 rows selected.
```

Practice 14 (continued)

12. Query the ALL_TABLES data dictionary view to see information about all the tables that you can access. Exclude tables that are owned by you.

```
TABLE_NAME   OWNER
----------   -----------
DEPT         <user2>
```

13. Revoke the SELECT privilege from the other team.

15

SQL Workshop

Workshop Overview

- Creating tables and sequences
- Modifying data in the tables
- Modifying a table definition
- Creating a view
- Writing scripts containing SQL and SQL*Plus commands
- Generating a simple report

Workshop Overview

This workshop has you build a set of database tables for a video application. Once you create the tables, you will insert, update, and delete records in a video store database, and generate a report. The database contains only the essential tables.

Note: If you want to build the tables, you can execute the *buildtab.sql* script in SQL*Plus. If you want to drop the tables, you can execute the *dropvid.sql* script in SQL*Plus. Then you can execute the *buildvid.sql* script in SQL*Plus to create and populate the tables. If you use the *buildvid.sql* to build and populate the tables, start the exercises from Practice #6b.

Practice 15

1. Create the tables based on the table instance charts below. Choose the appropriate datatypes and be sure to add integrity constraints.

   a. Table name: MEMBER

Column_ Name	MEMBER_ID	LAST_ NAME	FIRST_ NAME	ADDRESS	CITY	PHONE	JOIN_ DATE
Key Type	PK						
Null/ Unique	NN,U	NN					NN
Default Value							System Date
Data Type	Number	Varchar2	Varchar2	Varchar2	Varchar2	Varchar2	Date
Length	10	25	25	100	30	15	

   b. Table name: TITLE

Column_ Name	TITLE_ID	TITLE	DESCRIPTION	RATING	CATEGORY	RELEASE_ DATE
Key Type	PK					
Null/ Unique	NN,U	NN	NN			
Check					G, PG, R, NC17, NR	DRAMA, COMEDY, ACTION, CHILD, SCIFI, DOCUMENTARY
Data Type	Number	Varchar2	Varchar2	Varchar2	Varchar2	Date
Length	10	60	400	4	20	

Practice 15 (continued)

c. Table name: TITLE_COPY

Column Name	COPY_ID	TITLE_ID	STATUS
Key Type	PK	PK,FK	
Null/ Unique	NN,U	NN,U	NN
Check			AVAILABLE, DESTROYED, RENTED, RESERVED
Data Type	Number	Number	Varchar2
Length	10	10	15

d. Table name: RENTAL

Column Name	BOOK_ DATE	MEMBER_ ID	COPY_ ID	ACT_RET_ DATE	EXP_RET_ DATE	TITLE_ ID
Key Type	PK	PK,FK	PK,FK			PK,FK
Default Value	System Date				2 days	
FK Ref Table		member	copy_id			title_copy
FK Ref Col		member_id				title_id
Data Type	Date	Number	Number	Date	Date	Number
Length		10	10			10

Practice 15 (continued)

e. Table name: RESERVATION

Column_ Name	RES_ DATE	MEMBER_ ID	TITLE_ ID
Key Type	PK	PK,FK	PK,FK
Null/ Unique	NN,U	NN,U	NN
FK Ref Table		MEMBER	TITLE
FK Ref Column		member_id	title_id
Data Type	Date	Number	Number
Length		10	10

2. Verify that the tables and constraints were created properly by checking the data dictionary.

```
TABLE_NAME
-----------
MEMBER
RENTAL
RESERVATION
TITLE
TITLE_COPY
```

```
CONSTRAINT_NAME                C TABLE_NAME
------------------------------ - --------------
MEMBER_LAST_NAME_NN            C MEMBER
MEMBER_JOIN_DATE_NN            C MEMBER
MEMBER_MEMBER_ID_PK            P MEMBER
RENTAL_BOOK_DATE_COPY_TITLE_PK P RENTAL
RENTAL_MEMBER_ID_FK            R RENTAL
RENTAL_COPY_ID_TITLE_ID_FK     R RENTAL
RESERVATION_RESDATE_MEM_TIT_PK P RESERVATION
RESERVATION_MEMBER_ID          R RESERVATION
RESERVATION_TITLE_ID           R RESERVATION
...
18 rows selected.
```

Practice 15 (continued)

3. Create sequences to uniquely identify each row in the MEMBER table and the TITLE table.
   a. Member number for the MEMBER table: start with 101; do not allow caching of the values. Name the sequence member_id_seq.
   b. Title number for the TITLE table: start with 92; no caching. Name the sequence title_id_seq.
   c. Verify the existence of the sequences in the data dictionary.

```
SEQUENCE_NAME  INCREMENT_BY LAST_NUMBER
-------------- ------------ -----------
TITLE_ID_SEQ              1          92
MEMBER_ID_SEQ             1         101
```

4. Add data to the tables. Create a script for each set of data to add.
   a. Add movie titles to the TITLE table. Write a script to enter the movie information Save the script as *p15q4a.sql.* Use the sequences to uniquely identify each title. Remember that single quotation marks in a character field must be specially handled. Verify your additions.

```
TITLE
------------------------
Willie and Christmas Too
Alien Again
The Glob
My Day Off
Miracles on Ice
Soda Gang
6 rows selected.
```

Practice 15 (continued)

Title	Description	Rating	Category	Release_date
Willie and Christmas Too	All of Willie's friends made a Christmas list for Santa, but Willie has yet to add his own wish list.	G	CHILD	05-OCT-95
Alien Again	Yet another installation of science fiction history. Can the heroine save the planet from the alien life form?	R	SCIFI	19-MAY-95
The Glob	A meteor crashes near a small American town and unleashed carnivorous goo in this classic.	NR	SCIFI	12-AUG-95
My Day Off	With a little luck and a lot of ingenuity, a teenager skips school for a day in New York	PG	COMEDY	12-JUL-95
Miracles on Ice	A six-year-old has doubts about Santa Claus but she discovers that miracles really do exist.	PG	DRAMA	12-SEP-95
Soda Gang	After discovering a cache of drugs, a young couple find themselves pitted against a vicious gang.	NR	ACTION	01-JUN-95

Practice 15 (continued)

b. Add data to the MEMBER table. Write a script named *p15q4b.sql* to prompt users for the information. Execute the script. Be sure to use the sequence to add the member numbers.

First Name	**Last Name**	**Address**	**State**	**Phone**	**Join Date**
Carmen	Velasquez	283 King Street	Seattle	206-899-6666	08-MAR-90
LaDoris	Ngao	5 Modrany	Bratislava	586-355-8882	08-MAR-90
Midori	Nagayama	68 Via Centrale	Sao Paolo	254-852-5764	17-JUN-91
Mark	Lewis	6921 King Way	Lagos	63-559-7777	07-APR-90
Audry	Ropeburn	86 Chu Street	Hong Kong	41-559-87	18-JAN-91
Molly	Urguhart	3035 Laurier	Quebec	418-542-9988	18-JAN-91

Practice 15 (continued)

c. Add the following movie copies in the TITLE_COPY table:

Title	Copy Number	Status
Willie and Christmas Too	1	Available
Alien	1	Available
	2	Rented
The Glob	1	Available
My Day Off	1	Available
	2	Available
	3	Rented
Miracles on Ice	1	Available
Soda Gang	1	Available

d. Add the following rentals to the RENTAL table:
Note: Title number may be different depending on sequence number.

Title	Copy_ number	Customer	Date_ Rented	Date_return_expected	Date_ returned
92	1	101	3 days ago	1 day ago	2 days ago
93	2	101	1 day ago	1 day from now	
95	3	102	2 days ago	Today	
97	1	106	4 days ago	2 days ago	2 days ago

5. Create a view named TITLE_AVAIL to show the movie titles and the availability of each copy and its expected return date if rented. Query all rows from the view.

```
TITLE                COPY_ID STATUS     EXP_RET_D
------------------ ------- ---------- ------------
Alien Again                1 AVAILABLE
Alien Again                2 RENTED     05-NOV-97
Miracles on Ice            1 AVAILABLE
My Day Off                 1 AVAILABLE
My Day Off                 2 AVAILABLE
My Day Off                 3 RENTED     06-NOV-97
Soda Gang                  1 AVAILABLE  04-NOV-97
The Glob                   1 AVAILABLE
Willie and Christmas Too 1 AVAILABLE  05-NOV-97
9 rows selected.
```

6. Make changes to data in the tables.

   a. Add a new title. The movie is “Interstellar Wars,” which is rated PG and classified as a Sci-fi movie. The release date is 07-JUL-77. The description is “Futuristic interstellar action movie. Can the rebels save the humans from the evil Empire?” Be sure to add a title copy record for two copies.

   b. Enter two reservations. One reservation is for Carmen Velasquez, who wants to rent “Interstellar Wars.” The other is for Mark Lewis, who wants to rent “Soda Gang.”

c. Customer Carmen Velasquez rents the movie "Interstellar Wars," copy 1. Remove her reservation for the movie. Record the information about the rental. Allow the default value for the expected return date to be used. Verify that the rental was recorded by using the view you created.

```
TITLE                     COPY_ID STATUS    EXP_RET_D
------------------------- ------- --------- ----------------
Alien Again                     1 AVAILABLE
Alien Again                     2 RENTED    05-NOV-97
Interstellar Wars               1 RENTED    08-NOV-97
Interstellar Wars               2 AVAILABLE
Miracles on Ice                 1 AVAILABLE
My Day Off                      1 AVAILABLE
My Day Off                      2 AVAILABLE
My Day Off                      3 RENTED    06-NOV-97
Soda Gang                       1 AVAILABLE 04-NOV-97
The Glob                        1 AVAILABLE
Willie and Christmas Too        1 AVAILABLE 05-NOV-97
9 rows selected.
```

7. Make a modification to one of the tables.

a. Add a PRICE column to the TITLE table to record the purchase price of the video. The column should have a total length of eight digits and two decimal places. Verify your modifications.

```
 Name           Null?      Type
-------------- ---------  -----
TITLE_ID       NOT NULL   NUMBER(10)
TITLE          NOT NULL   VARCHAR2(60)
DESCRIPTION    NOT NULL   VARCHAR2(400)
RATING                    VARCHAR2(4)
CATEGORY                  VARCHAR2(20)
RELEASE_DATE              DATE
PRICE                     NUMBER(8,2)
```

8. Create a report titled Customer History Report. This report will contain each customer's history of renting videos. Be sure to include the customer name, movie rented, dates of the rental, and duration of rentals. Total the number of rentals for all customers for the reporting period. Save the script in a file named *p15q8.sql.*

Practice 15 (continued)

b. Create a script named *p15q7b.sql* to update each video with a price according to the following list. Note: Have the title id numbers available for this exercise.

Title	Price
Willie and Christmas Too	25
Alien Again	35
The Glob	35
My Day Off	35
Miracles on Ice	98
Soda Gang	35
Interstellar Wars	29

c. Ensure that in the future all titles will contain a price value. Verify the constraint.

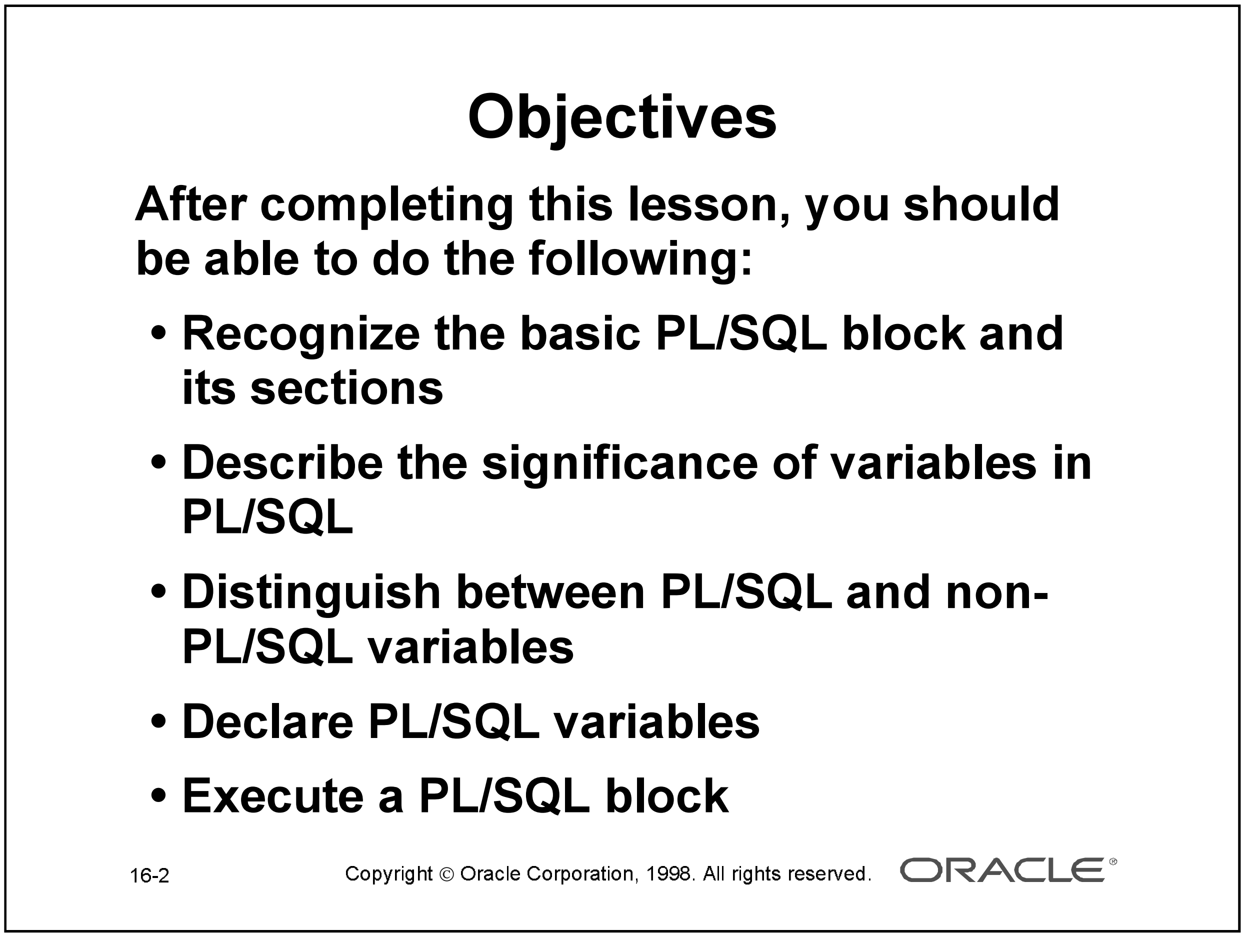
```
CONSTRAINT_NAME     C  SEARCH_CONDITIONS
------------------  -- -----------------
TITLE_PRICE_NN      C  PRICE IS NOT NULL
```

8. Create a report titles Customer History Report. This report will contain each customer's history of renting videos. Be sure to include the customer name, movie rented, dates of the rental, and duration of rentals. Total the number of rentals for all customers for the reporting period. Save the script in a file name *p15q8.sql.*

```
MEMBER            TITLE                 BOOK_DATE DURATION
--------------    ------------------    --------- --------
LaDoris Ngao      The Glob              04-NOV-97
Molly Urguhart    Miracles on Ice       02-NOV-97        2
Carmen Velasquez  Willie and Christmas  03-NOV-97        1
                  Too
                  Willie and Christmas  03-NOV-97        1
                  Too
                  Alien Again           05-NOV-97
```

16

Declaring Variables



Objectives

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

- Recognize the basic PL/SQL block and its sections
- Describe the significance of variables in PL/SQL
- Distinguish between PL/SQL and non-PL/SQL variables
- Declare PL/SQL variables
- Execute a PL/SQL block

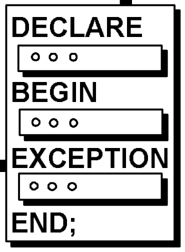
16-2

Lesson Aim

This lesson presents the basic rules and structure for writing and executing PL/SQL blocks of code. It also shows you how to declare variables and and assign datatypes to them.

PL/SQL Block Structure

- DECLARE – Optional
  - Variables, cursors, user-defined exceptions
- BEGIN – Mandatory
  - SQL statements
  - PL/SQL statements
- EXCEPTION – Optional
  - Actions to perform when errors occur
- END; – Mandatory



PL/SQL Block Structure

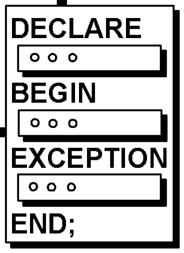
PL/SQL is a block-structured language, meaning that programs can be divided into logical blocks. A PL/SQL block is comprised of up to three sections: declarative (optional), executable (required), and exception handling (optional). Only BEGIN and END keywords are required. You can declare variables locally to the block that uses them. Error conditions (known as exceptions) can be handled specifically within the block to which they apply. You can store and change values within a PL/SQL block by declaring and referencing variables and other identifiers.

The following table describes the three block sections.

Section	Description	Inclusion
Declarative	Contains all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and declarative sections	Optional
Executable	Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block	Mandatory
Exception handling	Specifies the actions to perform when errors and abnormal conditions arise in the executable section	Optional

PL/SQL Block Structure

```
DECLARE
  v_variable  VARCHAR2(5);
BEGIN
  SELECT     column_name
    INTO     v_variable
    FROM     table_name;
EXCEPTION
  WHEN exception_name THEN
  ...
END;
```



Executing Statements and PL/SQL Blocks from SQL*Plus

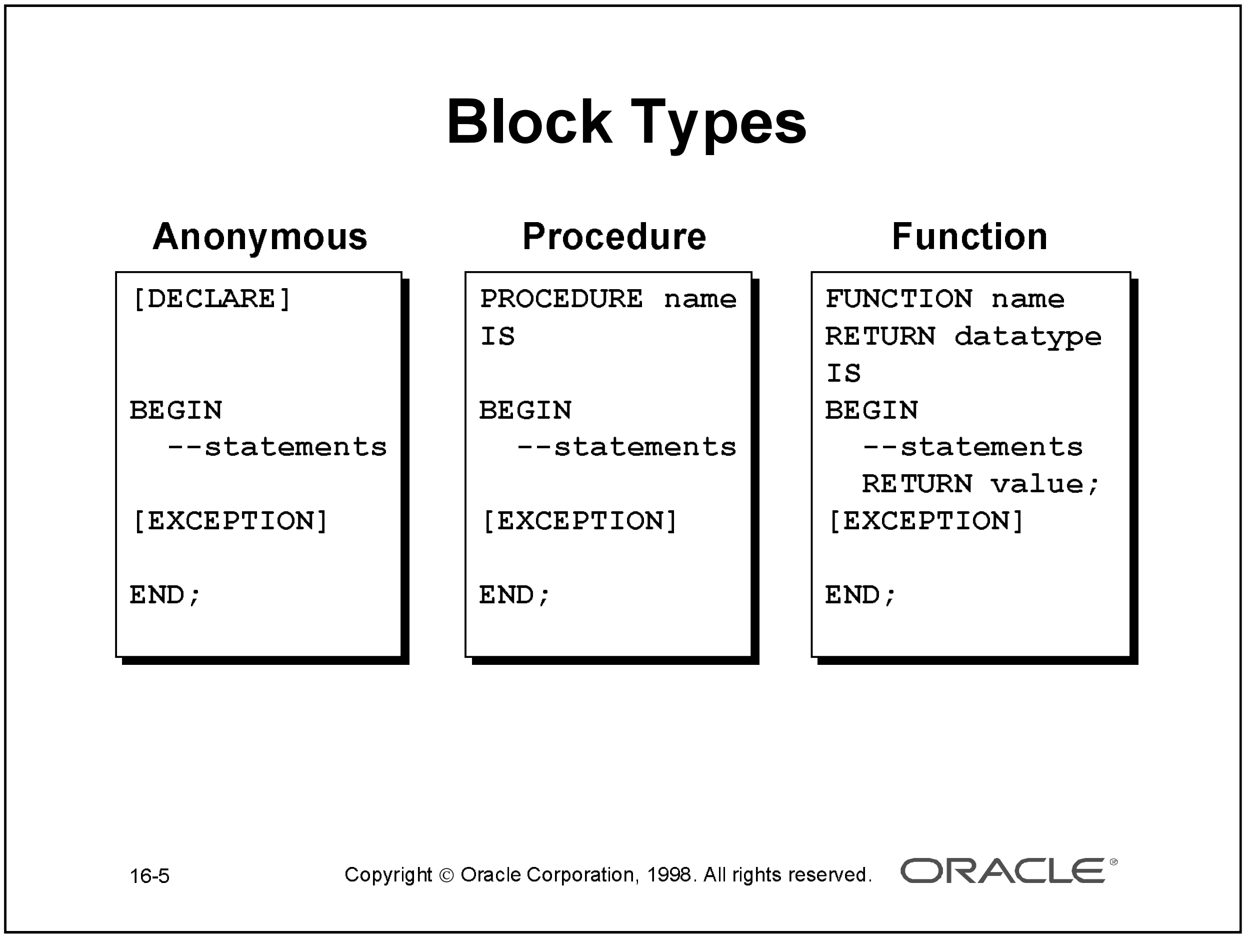
- Place a semicolon (;) at the end of a SQL statement or PL/SQL control statement.
- Place a forward slash (/) to run the anonymous PL/SQL block in the SQL buffer. When the block is executed successfully, without unhandled errors or compile errors, the message output should be as follows:

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

- Place a period (.) to close a SQL buffer. A PL/SQL block is treated as one continuous statement in the buffer, and the semicolons within the block do not close or run the buffer.

Note: In PL/SQL, an error is called an *exception.*

Section keywords DECLARE, BEGIN, and EXCEPTION are not followed by semicolons. However, END and all other PL/SQL statements do require a semicolon to terminate the statement. You can string statements together on the same line. However, this method is not recommended for clarity or editing.



Block Types

Anonymous

```
[DECLARE]


BEGIN
  --statements

[EXCEPTION]

END;
```

Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

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

END;
```

Every unit of PL/SQL comprises one or more blocks. These blocks can be entirely separate or nested one within another. The basic units (procedures and functions, also known as subprograms, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested subblocks. Therefore one block can represent a small part of another block, which in turn can be part of the whole unit of code. Of the two types of PL/SQL constructs available, anonymous blocks and subprograms, only anonymous blocks are covered in this course.

Anonymous Blocks

Anonymous blocks are unnamed blocks. They are declared at the point in an application where they are to be executed and are passed to the PL/SQL engine for execution at runtime. You can embed an anonymous block within a precompiler program and within SQL*Plus or Server Manager. Triggers in Developer/2000 components consist of such blocks.

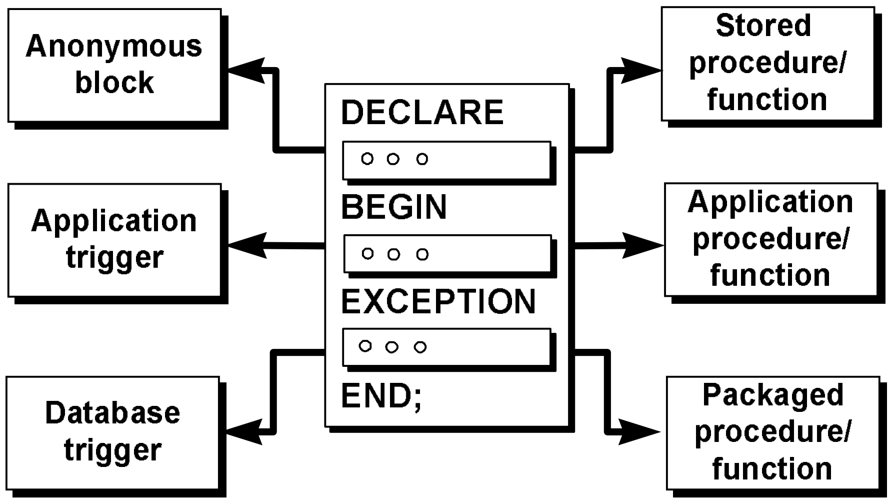
Subprograms

Subprograms are named PL/SQL blocks that can take parameters and can be invoked. You can declare them either as procedures or as functions. Generally you use a procedure to perform an action and a function to compute a value.

You can store subprograms at the server or application level. Using Developer/2000 components (Forms, Reports, and Graphics) you can declare procedures and functions as part of the application (a form or report), and call them from other procedures, functions, and triggers (see next page) within the same application whenever necessary.

Note: A function is similar to a procedure, except that a function *must* return a value. Procedures and functions are covered in the next PL/SQL course.

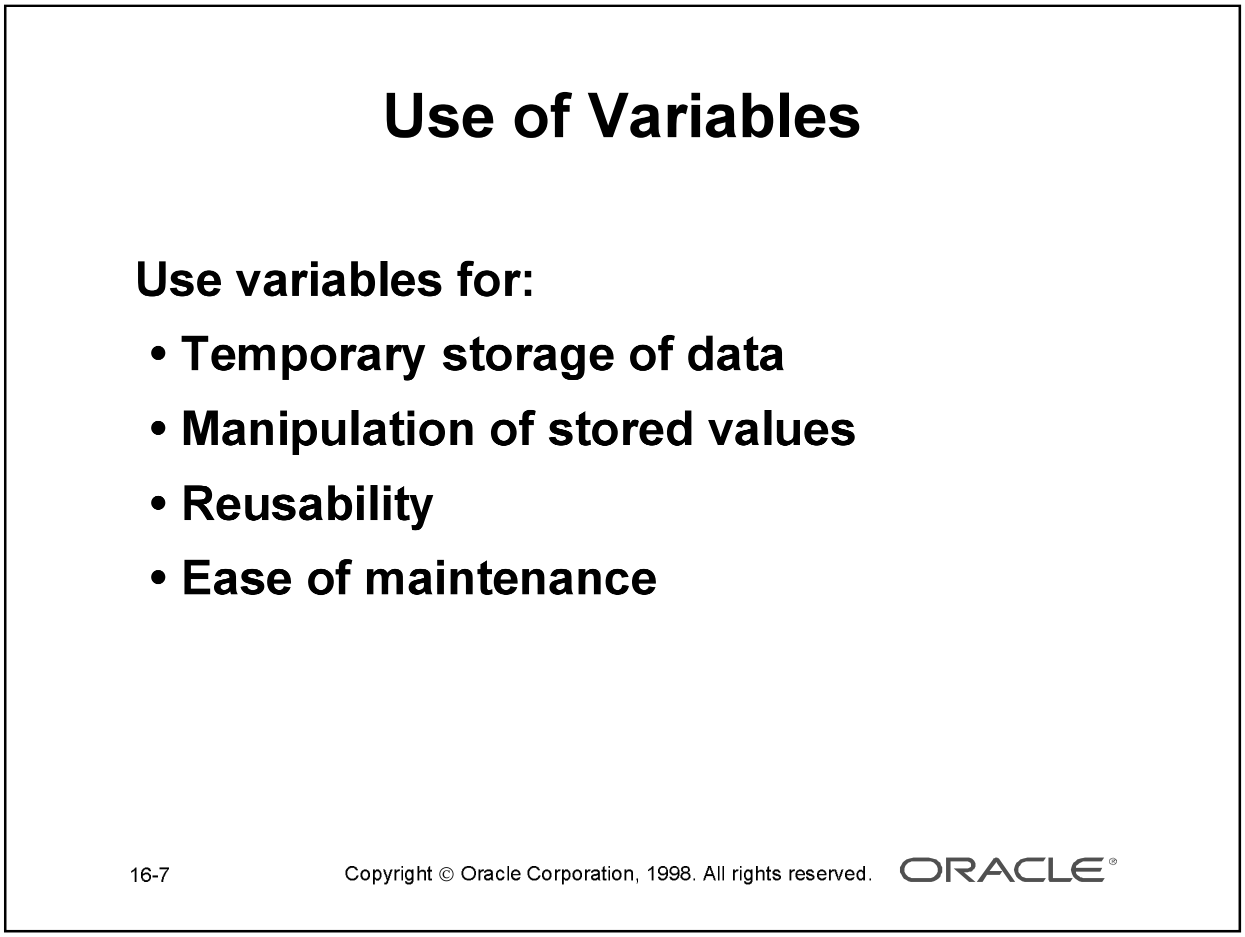
Program Constructs


Anonymous block
Application trigger
Database trigger
DECLARE
BEGIN
EXCEPTION
END;
Stored procedure/ function
Application procedure/ function
Packaged procedure/ function

16-6

The following table outlines a variety of different PL/SQL program constructs that use the basic PL/SQL block. They are available based on the environment where they are executed.

Program Construct	**Description**	**Availability**
Anonymous block	Unnamed PL/SQL block that is embedded within an application or is issued interactively	All PL/SQL environments
Stored procedure or function	Named PL/SQL block stored within the Oracle Server that can accept parameters and can be invoked repeatedly by name	Oracle Server
Application procedure or function	Named PL/SQL block stored within a Developer/2000 application or shared library that can accept parameters and can be invoked repeatedly by name	Developer/2000 components—for example, Forms
Package	Named PL/SQL module that groups together related procedures, functions, and identifiers	Oracle Server and Developer/2000 components—for example, Forms
Database trigger	PL/SQL block that is associated with a database table and is fired automatically when triggered by DML statements	Oracle Server
Application trigger	PL/SQL block that is associated with an application event and is fired automatically	Developer/2000 components—for example, Forms



Use of Variables

Use variables for:

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

16-7

Use of Variables

With PL/SQL you can declare variables then use them in SQL and procedural statements anywhere an expression can be used.

- Temporary storage of data

  Data can be temporarily stored in one or more variables for use when validating data input for processing later in data flow process.

- Manipulation of stored values

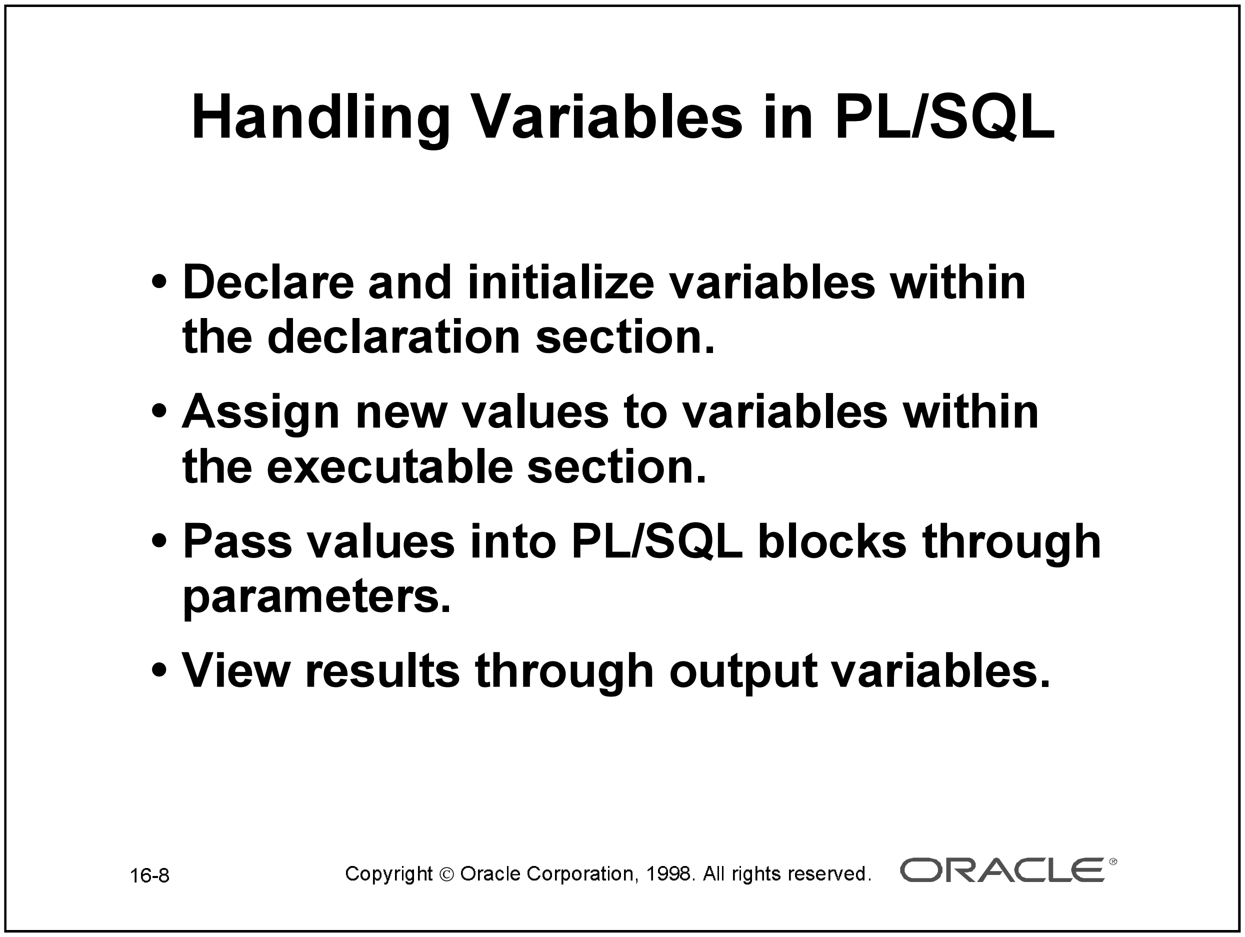
  Variables can be used for calculations and other data manipulations without accessing the database.

- Reusability

  Once declared, variables can be used repeatedly within an application simply by referencing them in other statements, including other declarative statements.

- Ease of maintenance

  When using %TYPE and %ROWTYPE (more information on %ROWTYPE is covered in a subsequent lesson) you declare variables, basing the declarations on the definitions of database columns. PL/SQL variables or cursor variables previously declared within the current scope may also use the %TYPE and %ROWTYPE attributes as datatype specifiers. If an underlying definition changes, the variable declaration changes accordingly at runtime. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.



Handling Variables in PL/SQL

- Declare and initialize variables within the declaration section.
- Assign new values to variables within the executable section.
- Pass values into PL/SQL blocks through parameters.
- View results through output variables.

16-8 ORACLE®

Handling Variables in PL/SQL

- Declare and initialize variables within the declaration section.

  You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the NOT NULL constraint.

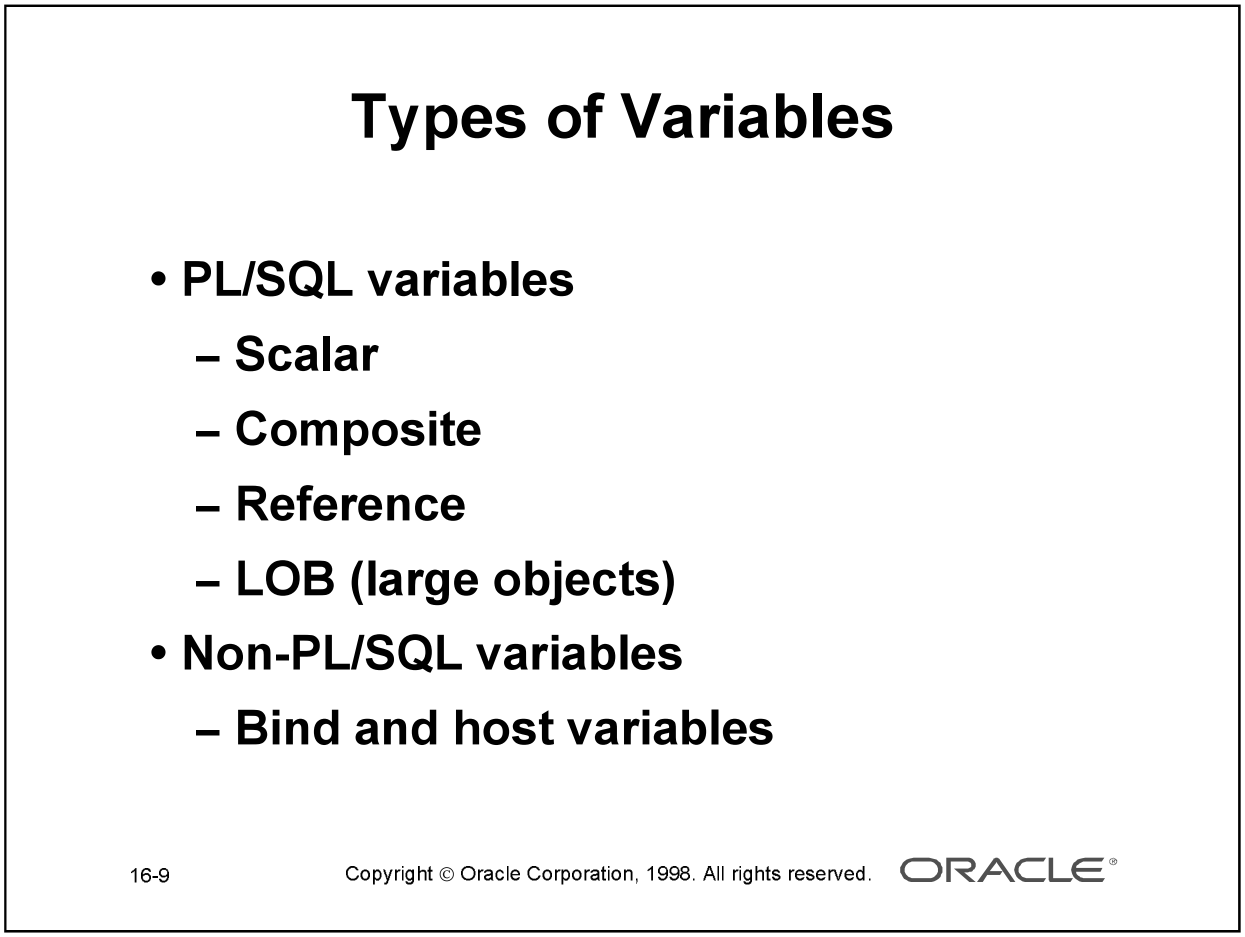
- Assign new values to variables within the executable section.
  - The existing value of the variable is replace with a new one.
  - Forward references are not allowed. You must declare a variable before referencing it in other statements, including other declarative statements.

- Pass values into PL/SQL subprograms through parameters.

  There are three parameter modes, IN (the default), OUT, and IN OUT. You use the IN parameter to pass values to the subprogram being called. You use the OUT parameter to return values to the caller of a subprogram. And you use the IN OUT parameter to pass initial values to the subprogram being called and to return updated values to the caller. IN and OUT subprogram parameters are covered in the PL/SQL Program Units course.

- View the results from a PL/SQL block through output variables.

  You can use reference variables for input or output in SQL data manipulation statements.



Types of Variables

- PL/SQL variables
  - Scalar
  - Composite
  - Reference
  - LOB (large objects)
- Non-PL/SQL variables
  - Bind and host variables

16-9 ORACLE®

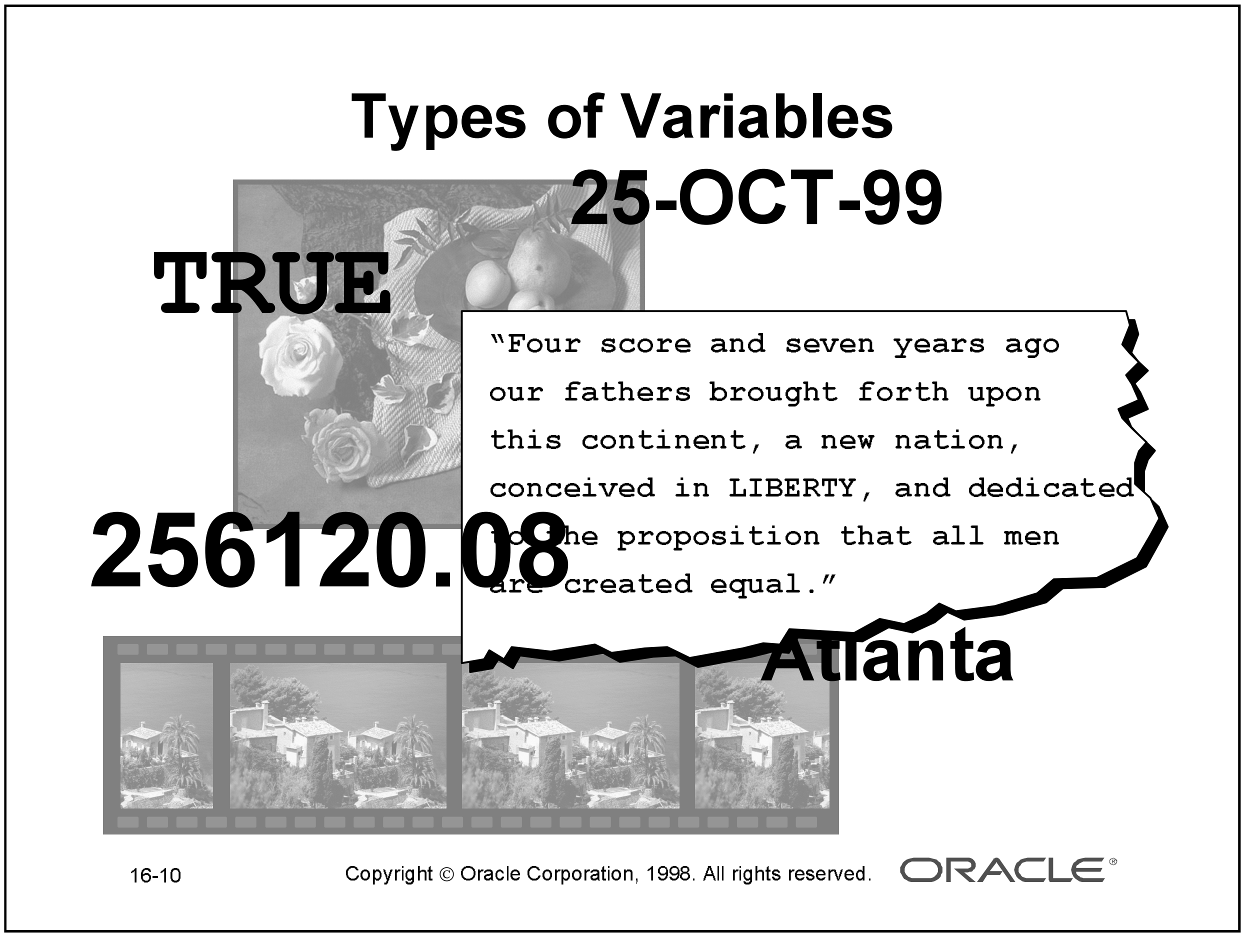
All PL/SQL variables have a datatype, which specifies a storage format, constraints, and valid range of values. PL/SQL supports four datatype categories—scalar, composite, reference, and LOB (large object)—that you can use for declaring variables, constants, and pointers.

- Scalar datatypes hold a single value. The main datatypes are those that correspond to column types in Oracle Server tables; PL/SQL also supports Boolean variables.
- Composite datatypes such as records allow groups of fields to be defined and manipulated in PL/SQL blocks. Composite datatypes are only briefly mentioned in this course.
- Reference datatypes hold values, called *pointers,* that designate other program items. Reference datatypes are not covered in this course.
- LOB datatypes hold values, called *locators,* that specify the location of large objects (graphic images for example) that are stored out of line. LOB datatypes are only briefly mentioned in this course.

Non-PL/SQL variables include host language variables declared in precompiler programs, screen fields in Forms applications, and SQL*Plus host variables.

Substitution variables in SQL*Plus allow portions of command syntax to be stored and then edited into the command before it is run. These are true host variables in that you can use them to pass runtime values, number or character, into or out of a PL/SQL block. You can then reference them within a PL/SQL block as host variables with a preceding colon.

For more information on LOBs, see
PL/SQL User's Guide and Reference, Release 8, "Fundamentals."



Types of Variables

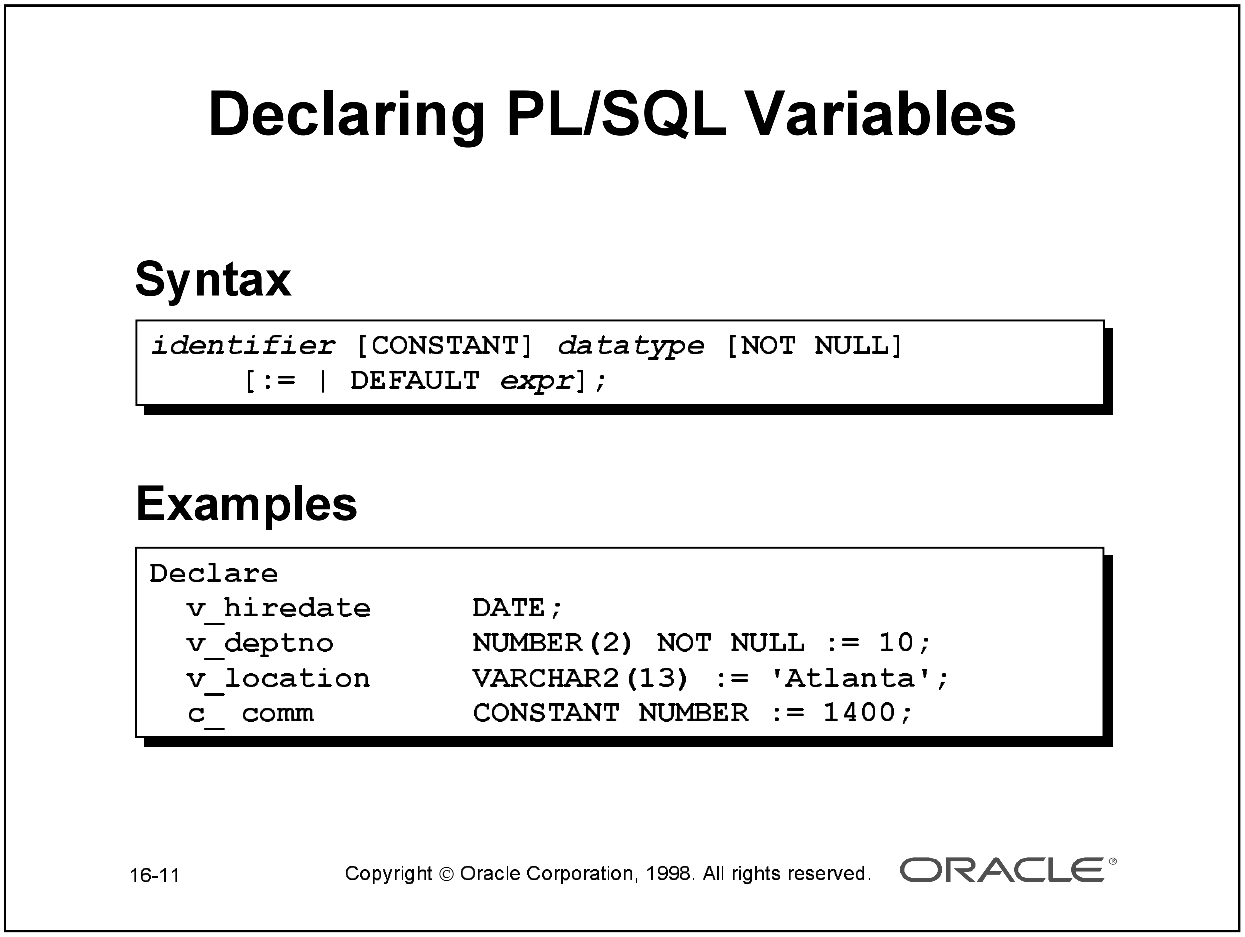
25-OCT-99
TRUE
"Four score and seven years ago our fathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."
256120.08
Atlanta

16-10

Types of Variables

Examples of some of the different variable datatypes in the above illustration are as follows.

- TRUE represents a Boolean value.
- 25-OCT-99 represents a DATE.
- The photograph represents a BLOB.
- The text of a speech represents a LONG RAW.
- 256120.08 represents a NUMBER datatype with precision.
- The movie represents a BFILE.
- The city name represents a VARCHAR2.



Declaring PL/SQL Variables

Syntax

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

Examples

```
Declare
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_ comm         CONSTANT NUMBER := 1400;
```

16-11

Declaring PL/SQL Variables

You need to declare all PL/SQL identifiers within the declaration section before referencing them within the PL/SQL block. You have the option to assign an initial value. You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, you must be sure to declare them separately in a previous statement.

In the syntax,

identifier	is the name of the variable.
CONSTANT	constrains the variable so that its value cannot change; constants must be initialized.
datatype	is a scalar, composite, reference, or LOB datatype (this course covers only scalar and composite datatypes).
NOT NULL	constrains the variable so that it must contain a value; NOT NULL variables must be initialized.
expr	is any PL/SQL expression that can be a literal, another variable, or an expression involving operators and functions.

Declaring PL/SQL Variables

Guidelines

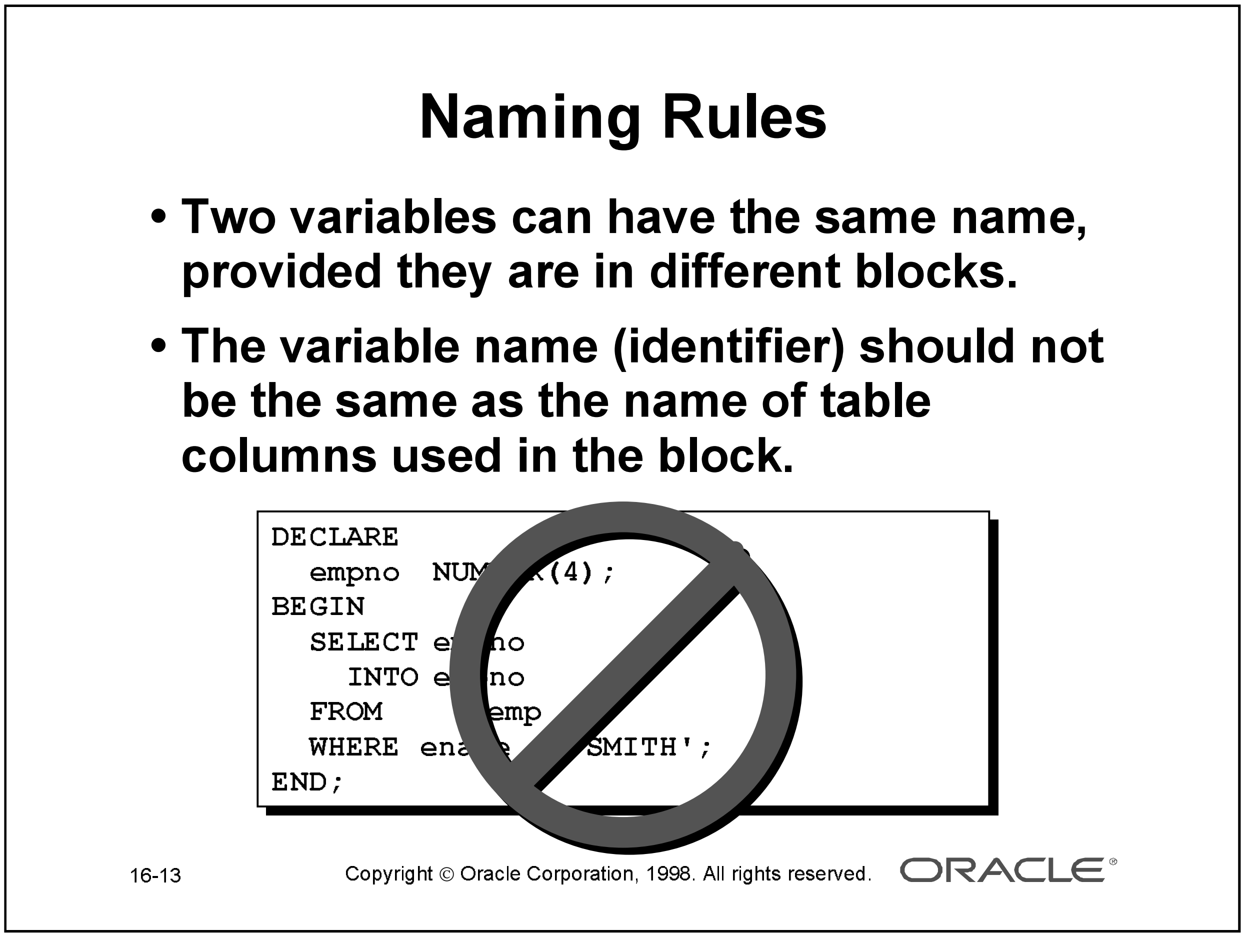
- **Follow naming conventions.**
- **Initialize variables designated as NOT NULL.**
- **Initialize identifiers by using the assignment operator (:=) or by using the DEFAULT reserved word.**
- **Declare at most one identifier per line.**

Guidelines

The assigned expression can be a literal, another variable, or an expression involving operators and functions.

- Name the identifier according to the same rules used for SQL objects.
- You can use naming conventions—for example, *v_name* to represent a variable and *c_name* to represent a constant variable.
- Initialize the variable to an expression with the assignment operator (:=) or, equivalently, with the DEFAULT reserved word. If you do not assign an initial value, the new variable contains NULL by default until you assign it later.
- If you use the NOT NULL constraint, you must assign a value.
- By declaring only one identifier per line code is more easily read and maintained.
- In constant declarations, the keyword CONSTANT must precede the type specifier. The following declaration names a constant of NUMBER sub-type REAL and assigns the value of 50000 to the constant. A constant must be initialized in its declaration; otherwise you get a compilation error when the declaration is elaborated (compiled).

```
v_sal     CONSTANT REAL := 50000.00;
```



Naming Rules

- **Two variables can have the same name, provided they are in different blocks.**
- **The variable name (identifier) should not be the same as the name of table columns used in the block.**

```
DECLARE
  empno  NUM   (4);
BEGIN
  SELECT e    no
    INTO e    no
  FROM        emp
  WHERE ena       SMITH';
END;
```

16-13

Naming Rules

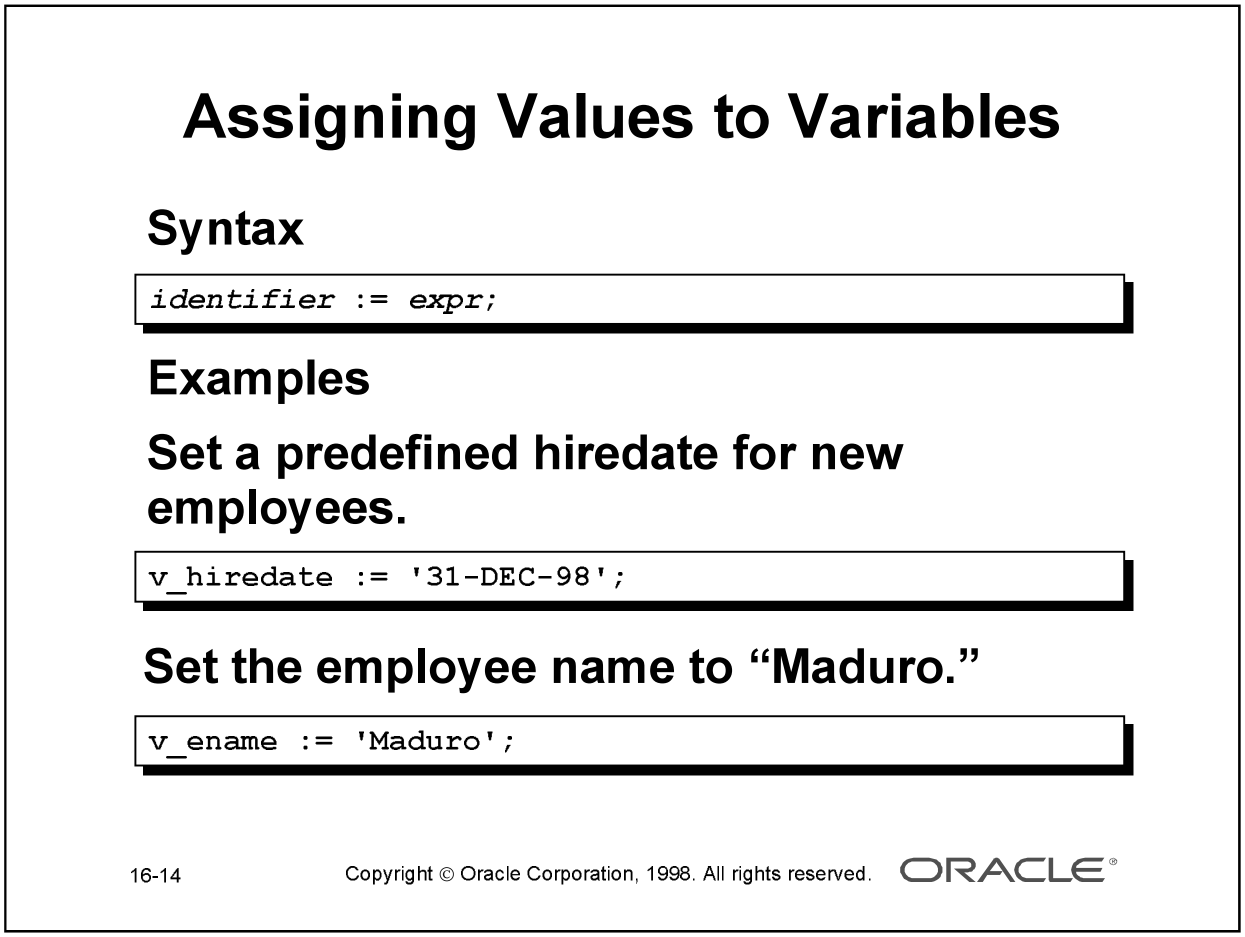
Two objects can have the same name, provided that they are defined in different blocks. Where they coexist, only the object declared in the current block can be used.

You should not choose the same name (identifier) for a variable as the name of table columns used in the block. If PL/SQL variables occur in SQL statements and have the same name as a column, the Oracle Server assumes that it is the column that is being referenced. Although the example code in the slide works; code written using the same name for a database table and variable name is not easy to read nor is it easy to maintain.

Consider adopting a naming convention for various objects such as the following example. Using v_ as a prefix representing *variable,* and g_ representing *global* variable, avoids naming conflicts with database objects.

```
DECLARE
   v_hiredate           date
   g_deptno             number(2) NOT NULL := 10;
BEGIN
...
```

Note: Identifiers must not be longer than 30 characters. The first character must be a letter; the remaining characters may be letters, numbers, or special symbols.



Assigning Values to Variables

Syntax

```
identifier := expr;
```

Examples

Set a predefined hiredate for new employees.

```
v_hiredate := '31-DEC-98';
```

Set the employee name to “Maduro.”

```
v_ename := 'Maduro';
```

16-14 Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

Assigning Values to Variables

To assign or reassign a value to a variable, you write a PL/SQL assignment statement. You must explicitly name the variable to receive the new value to the left of the assignment operator (:=).

In the syntax,

identifier	is the name of the scalar variable.
expr	can be a variable, literal, or function call, but *not* a database column.

The variable value assignment examples are defined as follows:

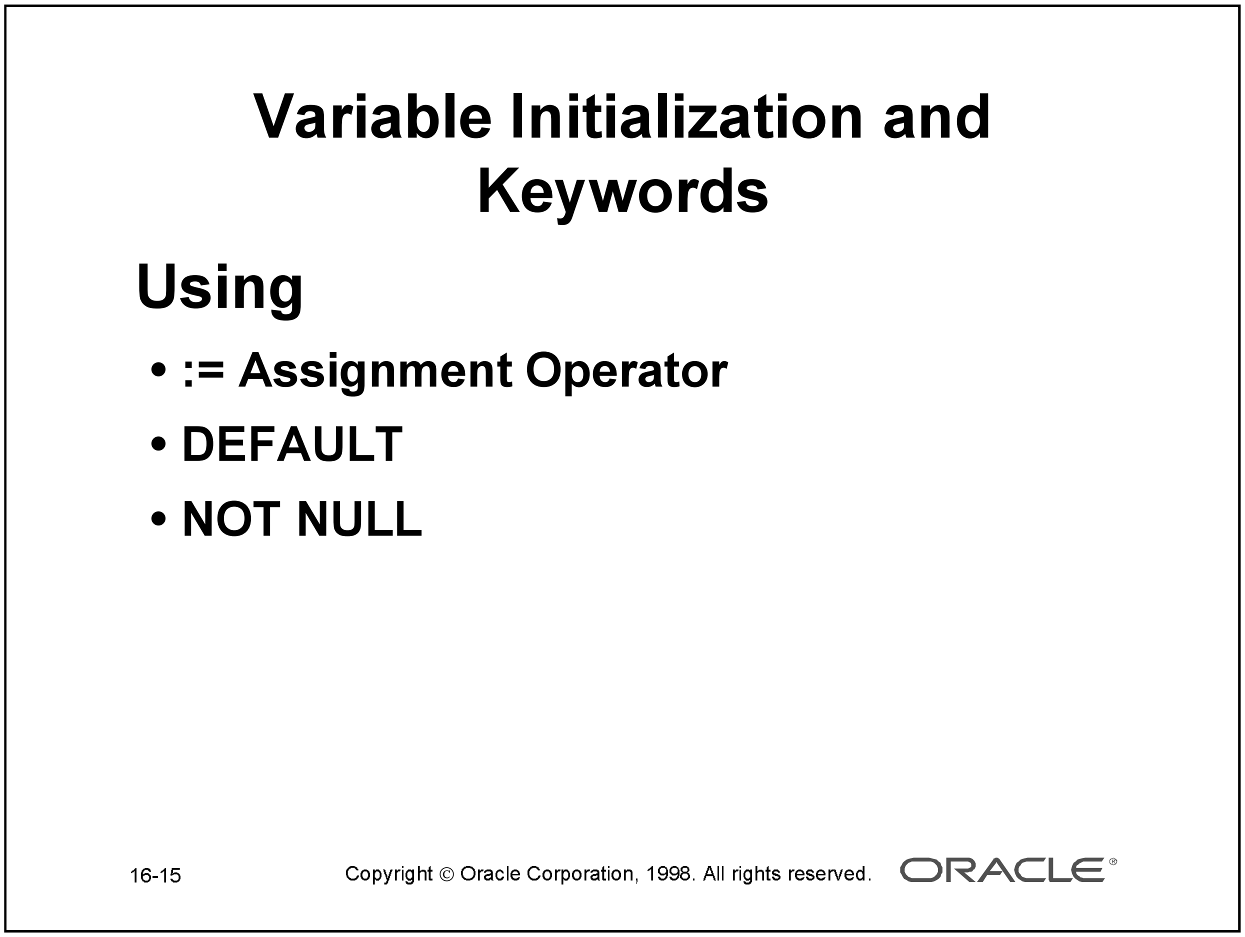
- Set the maximum salary identifier V_MAX_SAL to the value of current salary identifier V_SAL.
- Store the name “Maduro” in the v_ename identifier.

Another way to assign values to variables is to select or fetch database values into it. In the following example, you have Oracle compute a 10% bonus when you select the salary of an employee.

```
SQL> SELECT      sal * 0.10
  2  INTO        bonus
  3  FROM        emp
  4  WHERE       empno = 7369;
```

Then you can use the variable *bonus* in another computation or insert its value into a database table.

Note: To assign a value into a variable from the database, use a SELECT or FETCH statement.



Variable Initialization and Keywords

Using

- := Assignment Operator
- DEFAULT
- NOT NULL

Variables are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL. Unless you expressly initialize a variable, its value is undefined.

- Use the assignment (:=) operator for variables that have no typical value.

```
v_hiredate := to_date('15-SEP-99', 'DD-MON-YY');
```

Because the default date format set within the Oracle Server can differ from database to database, you may want to assign date values in a generic manner, as in the previous example.

- DEFAULT: You can use the keyword DEFAULT instead of the assignment operator to initialize variables. Use DEFAULT for variables that have a typical value.

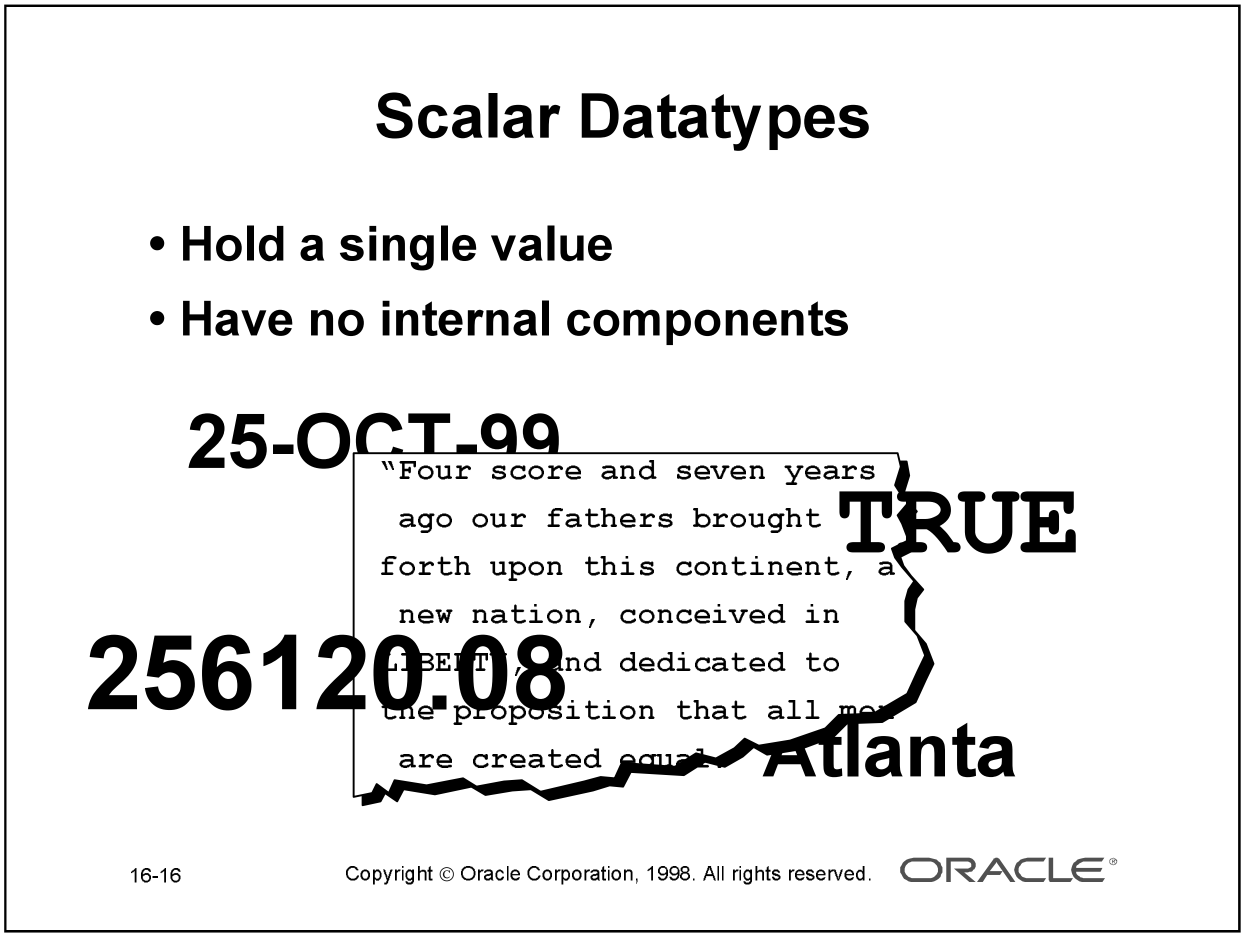
```
g_mgr       NUMBER(4) DEFAULT 7839;
```

- NOT NULL: Impose the NOT NULL constraint when the variable must contain a value.

  You cannot assign nulls to a variable defined as NOT NULL. The NOT NULL constraint must be followed by an initialization clause.

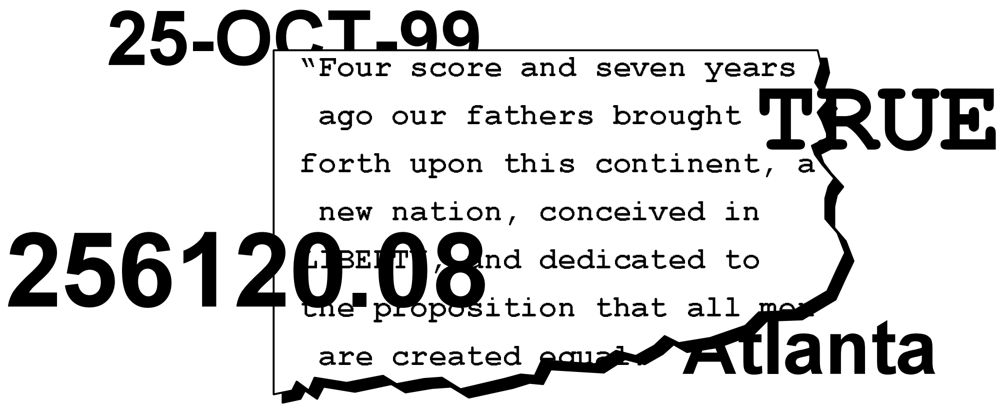
```
v_location  VARCHAR2(13) NOT NULL := 'CHICAGO';
```

Note: String literals must be enclosed in single quotation marks—for example, 'Hello, world'. If there is a single quotation mark in the string, write a single quotation mark twice—for example, 'Account wasn''t found'.



Scalar Datatypes

- Hold a single value
- Have no internal components

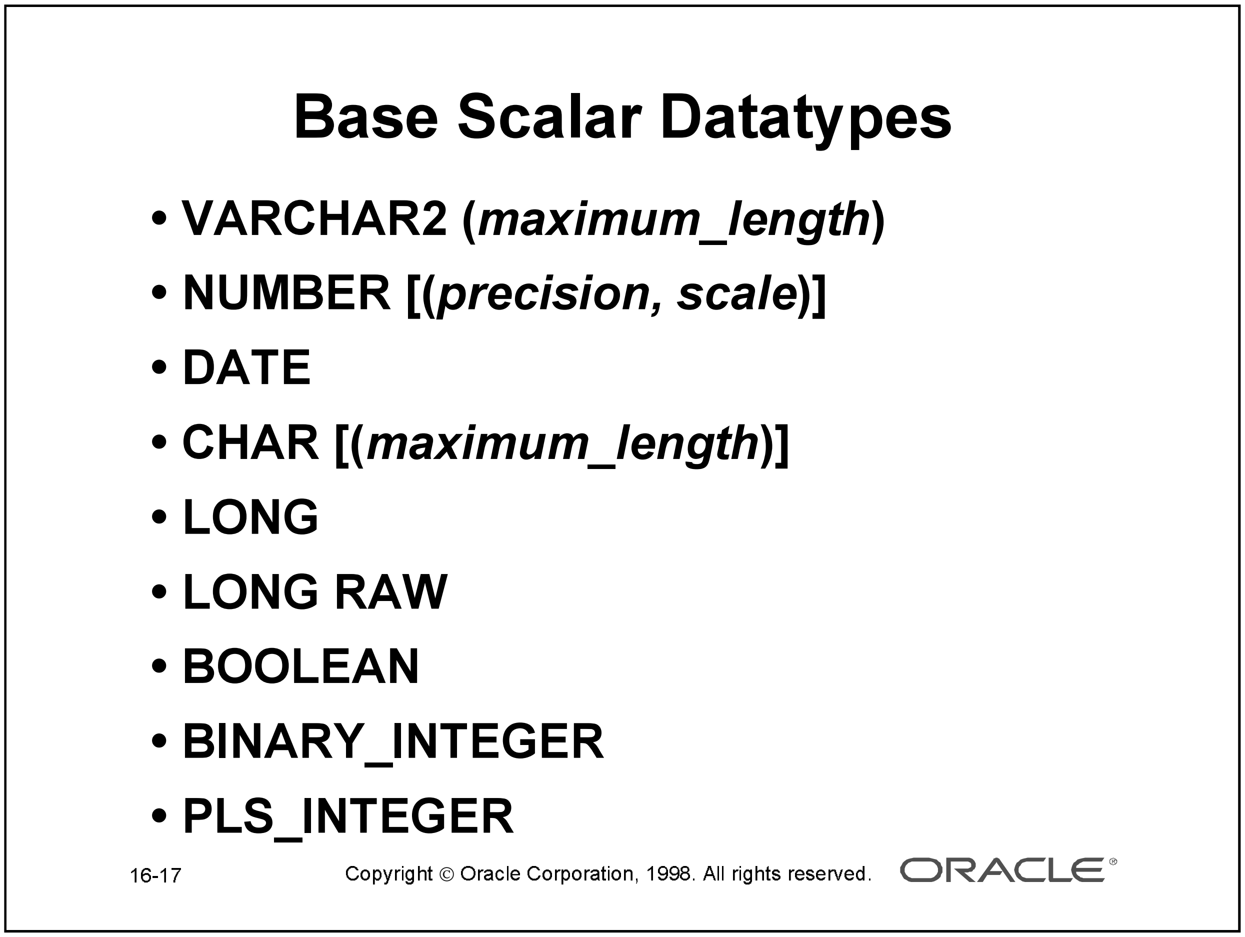

25-OCT-99
"Four score and seven years ago our fathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."
TRUE
256120.08
Atlanta

16-16

Scalar Datatypes

A scalar datatype holds a single value and has no internal components. Scalar datatypes can be classified into four categories: number, character, date, and Boolean. Character and number datatypes have subtypes that associate a base type to a constraint. For example, INTEGER and POSITIVE are subtypes of the NUMBER base type.

For more information and the complete list of scalar datatypes, see *PL/SQL User's Guide and Reference, Release 8,* "Fundamentals."



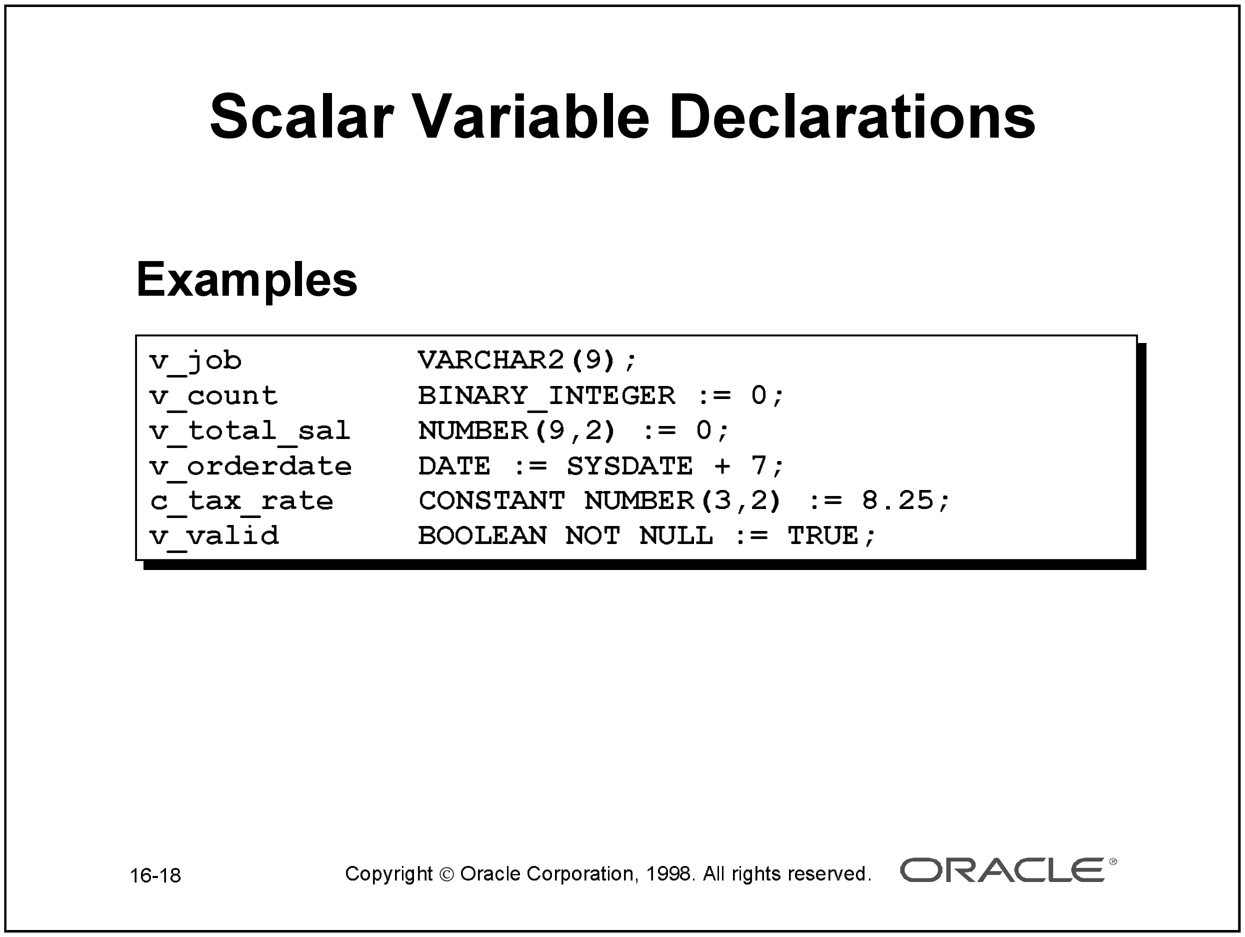
Base Scalar Datatypes

- **VARCHAR2 (*maximum_length*)**
- **NUMBER [(*precision, scale*)]**
- **DATE**
- **CHAR [(*maximum_length*)]**
- **LONG**
- **LONG RAW**
- **BOOLEAN**
- **BINARY_INTEGER**
- **PLS_INTEGER**

16-17

Datatype	**Description**
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(*precision, scale*)]	Base type for fixed and floating point numbers.
DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 4712 BC and 9999 AD.
CHAR [(*maximum_length*)]	Base type for fixed length character data up to 32767 bytes. If you do not specify a *maximum_length*, the default length is set to 1.
LONG	Base type for variable-length character data up to 32760 bytes. The maximum width of a LONG database column is 2147483647 bytes.
LONG RAW	Base type for binary data and byte strings up to 32760 bytes. LONG RAW data is not interpreted by PL/SQL.
BOOLEAN	Base type that stores one of three possible values used for logical calculations: TRUE, FALSE, or NULL.
BINARY_INTEGER	Base type for integers between -2147483647 and 2147483647.
PLS_INTEGER	Base type for signed integers between -2147483647 and 2147483647. PLS_INTEGER values require less storage and are faster than NUMBER and BINARY_INTEGER values.

Note: The LONG datatype is similar to VARCHAR2, except that the maximum length of a LONG value is 32,760 bytes. Therefore values longer than 32,760 bytes cannot be selected from a LONG database column into a LONG PL/SQL variable.



Scalar Variable Declarations

Examples

```
v_job         VARCHAR2(9);
v_count       BINARY_INTEGER := 0;
v_total_sal   NUMBER(9,2) := 0;
v_orderdate   DATE := SYSDATE + 7;
c_tax_rate    CONSTANT NUMBER(3,2) := 8.25;
v_valid       BOOLEAN NOT NULL := TRUE;
```

16-18

Declaring Scalar Variables

The variable declaration examples shown in the slide are defined as follows:

- Declared variable to store an employee job title.
- Declared variable to count the iterations of a loop and initialize the variable to 0.
- Declared variable to accumulate the total salary for a department and initialize the variable to 0.
- Declared variable to store the ship date of an order and initialize the variable to one week from today.
- Declared a constant variable for the tax rate, which never changes throughout the PL/SQL block.
- Declared flag to indicate whether a piece of data is valid or invalid and initialize the variable to TRUE.

The %TYPE Attribute

- **Declare a variable according to:**
  - **A database column definition**
  - **Another previously declared variable**
- **Prefix %TYPE with:**
  - **The database table and column**
  - **The previously declared variable name**

The %TYPE Attribute

When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct datatype and precision. If it is not, a PL/SQL error will occur during execution.

Rather than hard coding the datatype and precision of a variable, you can use the %TYPE attribute to declare a variable according to another previously declared variable or database column. The %TYPE attribute is most often used when the value stored in the variable will be derived from a table in the database or if the variable is destined to be written out to it. To use the attribute in place of the datatype required in the variable declaration, prefix it with the database table and column name. If referring to a previously declared variable, prefix the variable name to the attribute.

PL/SQL determines the datatype and size of the variable when the block is compiled, so it is always compatible with the column used to populate it. This is a definite advantage for writing and maintaining code, because there is no need to be concerned with column datatype changes made at the database level. You can also declare a variable according to another previously declared variable by prefixing the variable name to the attribute.

Declaring Variables with the %TYPE Attribute

Examples

```
...
  v_ename                emp.ename%TYPE;
  v_balance              NUMBER(7,2);
  v_min_balance          v_balance%TYPE := 10;
...
```

16-20

Declaring Variables with the %TYPE Attribute

Declare variables to store the name of an employee.

```
...
v_ename                  emp.ename%TYPE;
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10.

```
...
v_balance                NUMBER(7,2);
v_min_balance            v_balance%TYPE := 10;
...
```

A NOT NULL column constraint does not apply to variables declared using %TYPE. Therefore, if you declare a variable using the %TYPE attribute using a database column defined as NOT NULL, you can assign the NULL value to the variable.

Declaring BOOLEAN Variables

- Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.
- The variables are connected by the logical operators AND, OR, and NOT.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions may be used to return a Boolean value.

Declaring BOOLEAN Variables

With PL/SQL you can compare variables and in both SQL and procedural statements. These comparisons, called *Boolean expressions,* consist of simple or complex expressions separated by relational operators. In a SQL statement, you can use Boolean expressions to specify the rows in a table that are affected by the statement. In a procedural statement, Boolean expressions are the basis for conditional control.

NULL stands for a missing, inapplicable, or unknown value.

Examples

```
v_sal1 := 50000;
v_sal2 := 60000;
```

The following expression yields TRUE

```
v_sal1 < v_sal2
```

Declare and initialize a Boolean variable.

```
v_comm_sal BOOLEAN := (v_sal1 < v_sal2);
```

Composite Datatypes

Types

- PL/SQL TABLES
- PL/SQL RECORDS

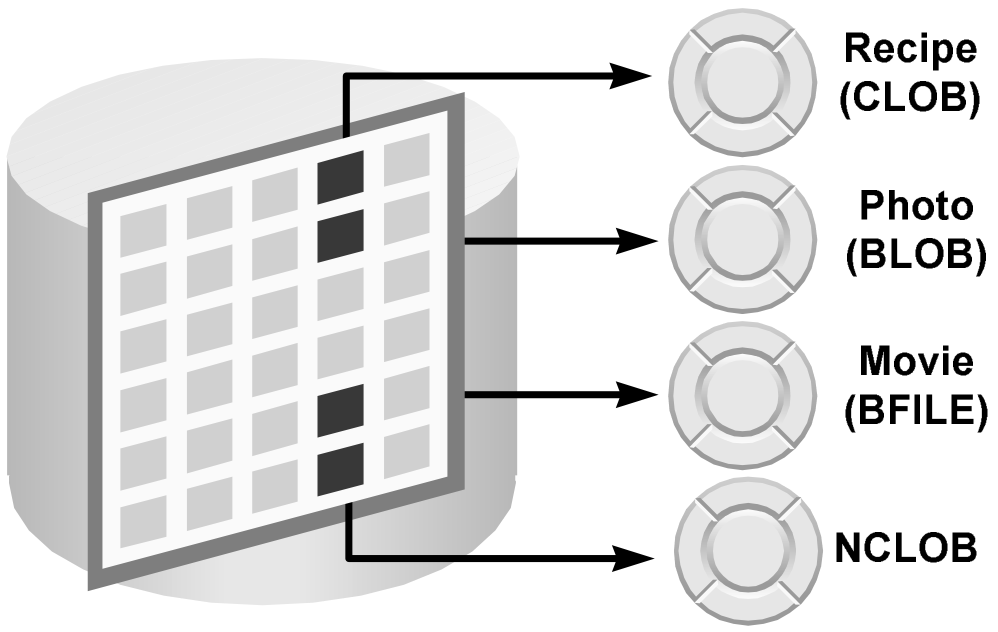
16-22

Composite Datatypes

Composite datatypes (also known as *collections*) are TABLE, RECORD, Nested TABLE, and VARRAY. You use the RECORD datatype to treat related but dissimilar data as a logical unit. You use the TABLE datatype to reference and manipulate collections of data as a whole object. Both RECORD and TABLE datatypes are covered in detail in a subsequent lesson. The Nested TABLE and VARRAY datatypes are not covered in this course.

For more information, see
PL/SQL User's Guide and Reference, Release 8, "Collections and Records."

LOB Datatype Variables



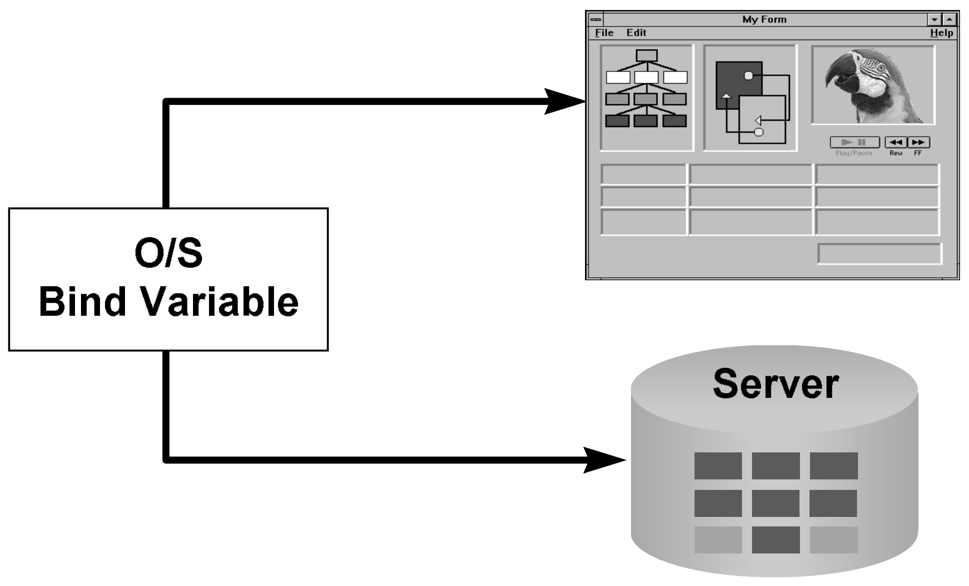
16-23 Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

LOB Datatype Variables

With the LOB (large object) Oracle8 datatypes you can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) up to 4 gigabytes in size. LOB datatypes allow efficient, random, piece-wise access to the data and can be attributes of an object type. LOBs also support random access to data.

- The CLOB (character large object) datatype is used to store large blocks of single-byte character data in the database.
- The BLOB (binary large object) datatype is used to store large binary objects in the database in line (inside the row) or out of line (outside the row).
- The BFILE (binary file) datatype is used to store large binary objects in operating system files outside the database.
- The NCLOB (national language character large object) datatype is used to store large blocks of single-byte or fixed-width multi-byte NCHAR data in the database, in line or out of line.

Bind Variables



A bind variable is a variable that you declare in a host environment, then use to pass runtime values, either number or character, into or out of one or more PL/SQL programs, which can use it like any other variable. You can reference variables declared in the host or calling environment in PL/SQL statements, unless the statement is within a procedure, function, or package. This includes host language variables declared in precompiler programs, screen fields in Developer/2000 Forms applications, and SQL*Plus bind variables.

Creating Bind Variables

In the SQL*Plus environment, to declare a bind variable, you use the command VARIABLE. For example, you declare a variable of type NUMBER as follows:

```
VARIABLE return_code NUMBER
```

Both SQL and SQL*Plus can reference the bind variable, and SQL*Plus can display its value.

Displaying Bind Variables

In the SQL*Plus environment, to display the current value of bind variables, you use the command PRINT. The following example illustrates a PRINT command:

```
SQL> VARIABLE n NUMBER
...
SQL> PRINT n
```

Referencing Non-PL/SQL Variables

Store the annual salary into a SQL*Plus host variable.

```
:g_monthly_sal := v_sal / 12;
```

- **Reference non-PL/SQL variables as host variables.**
- **Prefix the references with a colon (:).**

16-25

Assigning Values to Variables

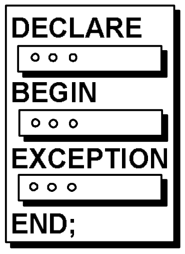
To reference host variables, you must prefix the references with a colon (:) to distinguish them from declared PL/SQL variables.

Examples

```
:host_var1 := v_sal;
:global_var1 := 'YES';
```

Summary

- PL/SQL blocks are composed of the following sections:
  - Declarative (optional)
  - Executable (required)
  - Exception handling (optional)
- A PL/SQL block can be an anonymous block, procedure, or function.



16-26

Summary

- PL/SQL identifiers:
  - Are defined in the declarative section
  - Can be of scalar, composite, reference, or LOB datatype
  - Can be based on the structure of another variable or database object
  - Can be initialized

16-27

Practice Overview

- Determining validity of declarations
- Developing a simple PL/SQL block

Practice Overview

This practice reinforces the basics of PL/SQL learned in this lesson, including data types, legal definitions of identifiers, and validation of expressions. You put all these elements together to create a simple PL/SQL block.

Paper-Based Questions

Questions 1 and 2 are paper-based questions.

Practice 16

Declare variables.

1. Evaluate each of the following declarations. Determine which of them are *not* legal and explain why.

a.

```
DECLARE
  v_id              NUMBER(4);
```

b.

```
DECLARE
  v_x, v_y, v_z     VARCHAR2(10);
```

c.

```
DECLARE
  v_birthdate       DATE NOT NULL;
```

d.

```
DECLARE
  v_in_stock        BOOLEAN := 1;
```

e.

```
DECLARE
  TYPE name_table_type IS TABLE OF VARCHAR2(20)
     INDEX BY BINARY_INTEGER;
  dept_name_table   name_table_type;
```

Practice 16 (continued)

2. In each of the following assignments, determine the data type of the resulting expression.

a.
```
v_days_to_go := v_due_date - SYSDATE;
```

____

b.
```
v_sender := USER || ': ' || TO_CHAR(v_dept_no);
```

____

c.
```
v_sum := $100,000 + $250,000;
```

____

d.
```
v_flag := TRUE;
```

____

e.
```
v_n1 := v_n2 > (2 * v_n3);
```

____

f.
```
v_value := NULL;
```

____

3. Create an anonymous block to output the phrase “My PL/SQL Block Works” to the screen.

```
G_MESSAGE
-----------------------
My PL/SQL Block Works
```

Practice 16 (continued)

If you have time, complete the following exercise.

4. Create a block that declares two variables. Assign the value of these PL/SQL variables to SQL*Plus host variables and print the results of the PL/SQL variables to the screen. Execute your PL/SQL block. Save your PL/SQL block to a file named *p16q4.sql*.

```
V_CHAR Character (variable length)
V_NUM  Number
```

Assign values to these variables as follows:

```
Variable Value
-------- -------------------------------------

V_CHAR   The literal '42 is the answer'
V_NUM    The first two characters from V_CHAR
```

```
G_CHAR
---------------------
42 is the answer


G_NUM
---------
42
```

17

Writing Executable Statements

Objectives

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

- **Recognize the significance of the executable section**
- **Write statements within the executable section**
- **Describe the rules of nested blocks**
- **Execute and test a PL/SQL block**
- **Use coding conventions**

Lesson Aim

In this lesson, you will learn how to write executable code within the PL/SQL block. You will also learn the rules for nesting PL/SQL blocks of code, as well as how to execute and test their PL/SQL code.

PL/SQL Block Syntax and Guidelines

- **Statements can continue over several lines.**
- **Lexical units can be separated by spaces:**
  - **Delimiters**
  - **Identifiers**
  - **Literals**
  - **Comments**

Because PL/SQL is an extension of SQL, the general syntax rules that apply to SQL are also applicable to the PL/SQL language.

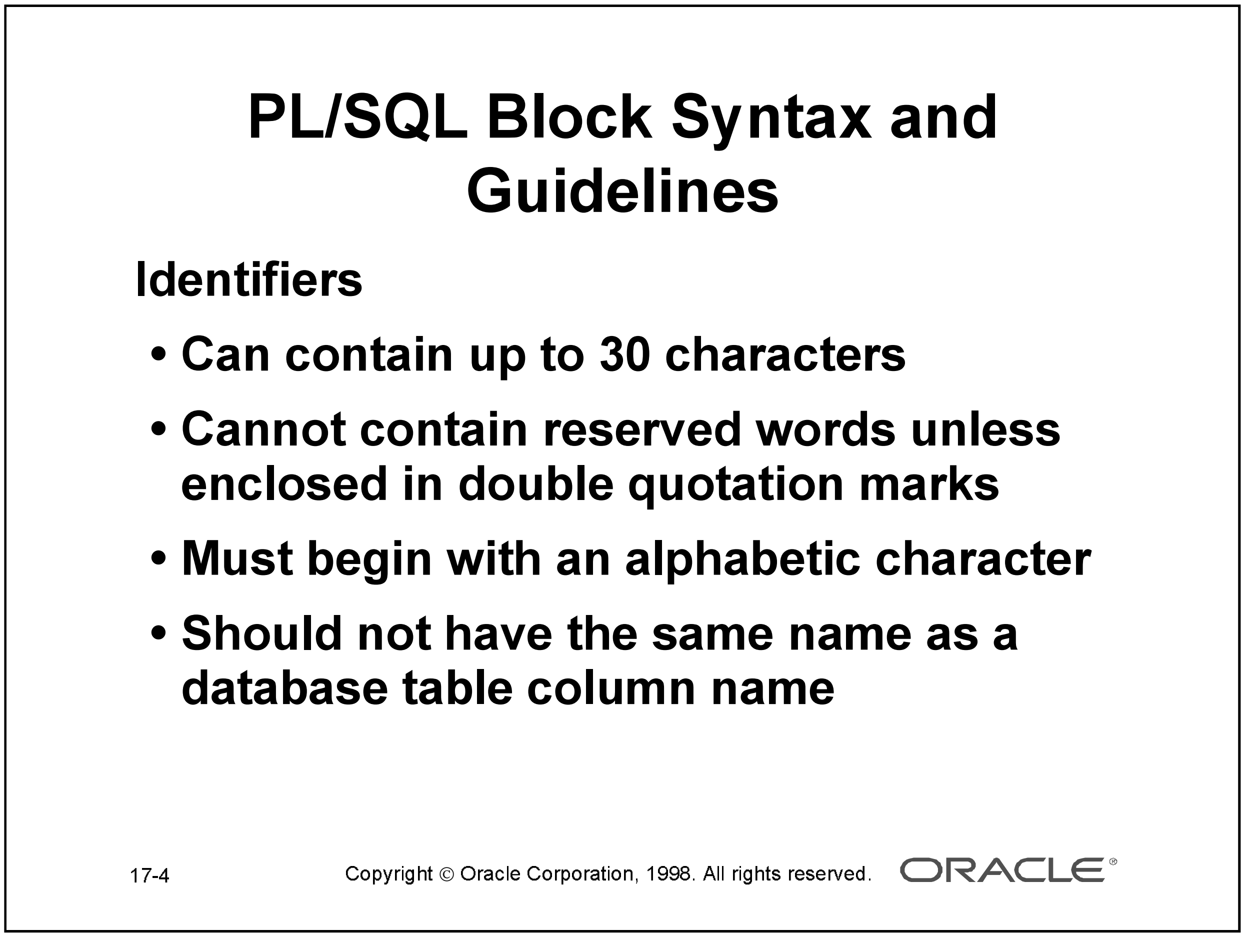
- Lexical units (for example, identifiers or literals) can be separated by one or more spaces or other delimiters that cannot be confused as being part of the lexical unit. You cannot embed spaces in lexical units except for string literals and comments.
- Statements can be split across lines, but keywords must not be split.

Delimiters

Delimiters are simple or compound symbols that have special meaning to PL/SQL.

Simple Symbols		**Compound Symbols**	
Symbol	**Meaning**	**Symbol**	**Meaning**
+	Addition Operator	<>	Relational Operator
-	Subtraction/Negation Operator	!=	Relational Operator
*	Multiplication Operator	||	Concatenation Operator
/	Division Operator	--	Single Line Comment Indicator
=	Relational Operator	/*	Beginning Comment Delimiter
@	Remote Access Indicator	*/	Ending Comment Delimiter
;	Statement Terminator	:=	Assignment Operator

For more information, see
PL/SQL User's Guide and Reference, Release 8, "Fundamentals."



PL/SQL Block Syntax and Guidelines

Identifiers

- **Can contain up to 30 characters**
- **Cannot contain reserved words unless enclosed in double quotation marks**
- **Must begin with an alphabetic character**
- **Should not have the same name as a database table column name**

17-4 Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

Identifiers

Identifiers are used to name PL/SQL program items and units, which include constants, variables, exceptions, cursors, cursor variables, subprograms, and packages.

- Identifiers can contain up to 30 characters, but they must start with an alphabetic character.
- Do not choose the same name for the identifier as the name of columns in a table used in the block. If PL/SQL identifiers are in the same SQL statements and have the same name as a column, then Oracle assumes that it is the column that is being referenced.
- Reserved words cannot be used as identifiers unless they are enclosed in double quotation marks (for example, "SELECT").
- Reserved words should be written in uppercase to promote readability.

For a complete list of reserved words, see
PL/SQL User's Guide and Reference, Release 8, "Appendix F."

PL/SQL Block Syntax and Guidelines

Literals

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

```
v_ename := 'Henderson';
```

- **Numbers can be simple values or scientific notation.**

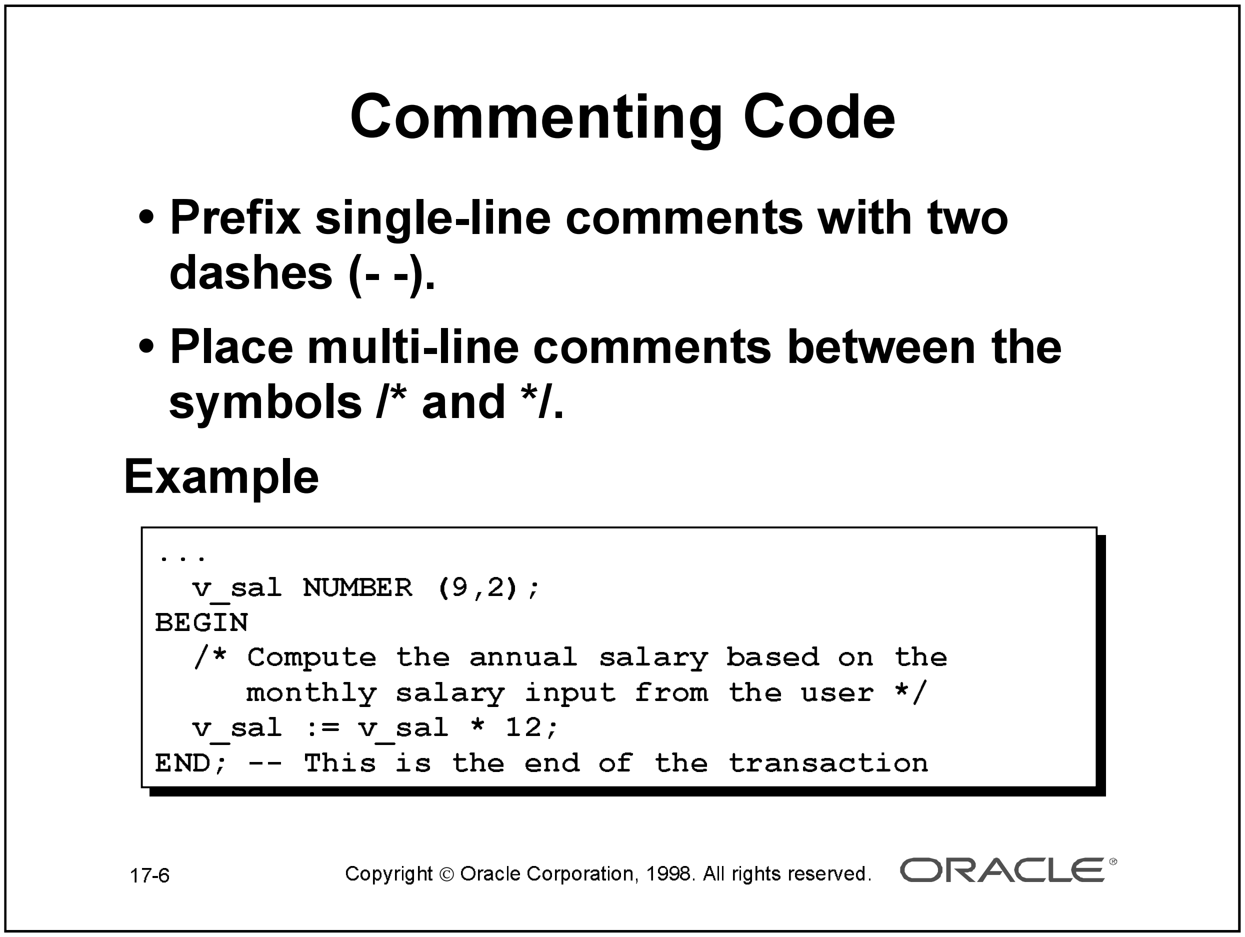
17-5 ORACLE®

Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier.

- Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.
- Numeric literals can be represented by either a simple value (for example, -32.5) or by scientific notation (for example, 2E5, meaning 2*10 to the power of 5 = 200000).

.



Commenting Code

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

Example

```
...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_sal := v_sal * 12;
END; -- This is the end of the transaction
```

17-6

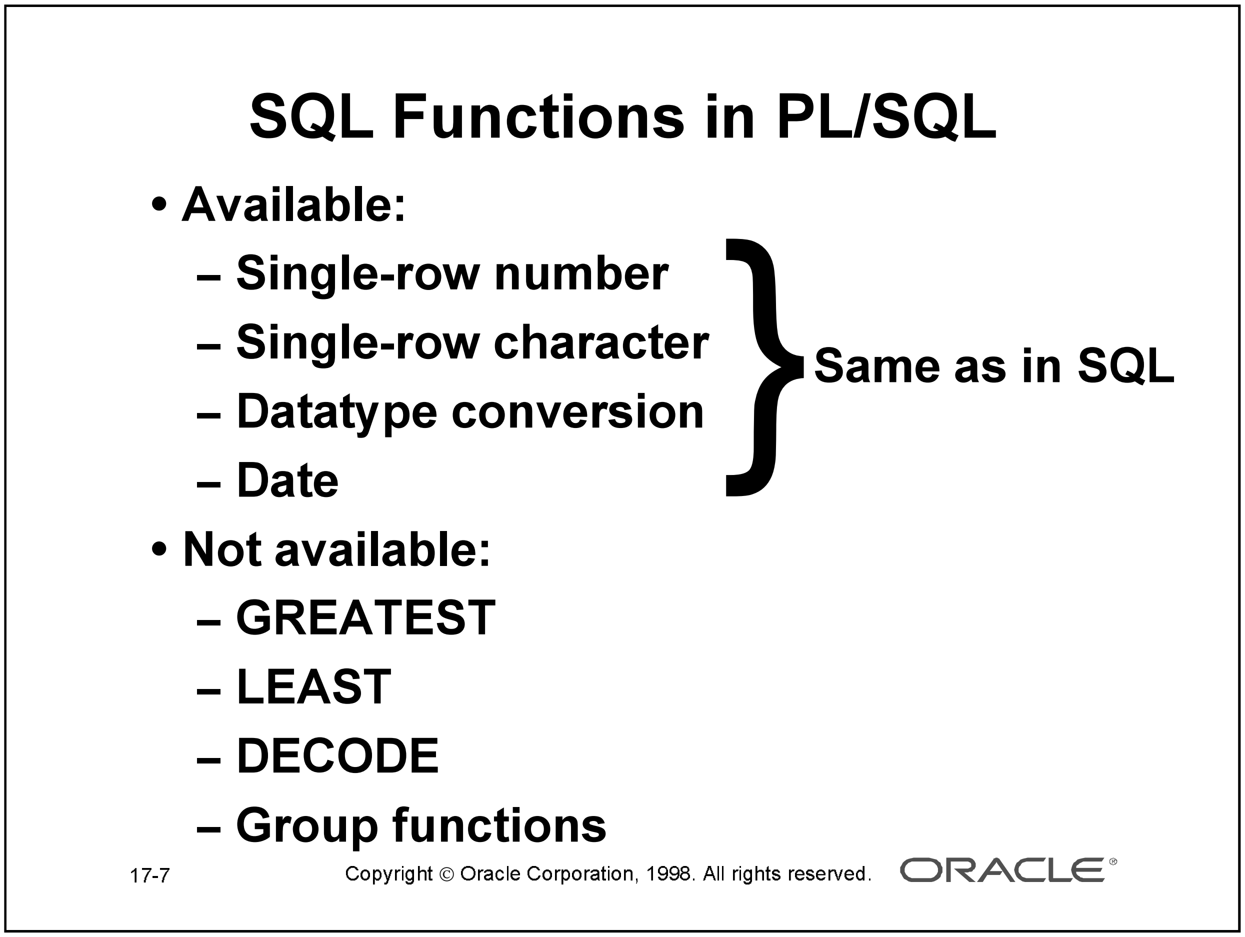
Commenting Code

Comment code to document each phase and to assist with debugging. Comment the PL/SQL code with two dashes (--) if the comment is on a single line, or enclose the comment between the symbols /* and */ if the comment spans several lines. Comments are strictly informational and do not enforce any conditions or behavior on behavioral logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance.

Example

Compute the yearly salary from the monthly salary.

```
...
  v_sal NUMBER(9,2);
BEGIN
  /* Compute the annual salary based on
  the monthly salary input from the user */
  v_sal := v_sal*12;
END;  -- This is the end of the transaction
```



SQL Functions in PL/SQL

- Available:
  - Single-row number
  - Single-row character
  - Datatype conversion
  - Date

  } Same as in SQL
- Not available:
  - GREATEST
  - LEAST
  - DECODE
  - Group functions

17-7

SQL Functions in PL/SQL

Most of the functions available in SQL are also valid in PL/SQL expressions:

- Single-row number functions
- Single-row character functions
- Datatype conversion functions
- Date functions
- Miscellaneous functions

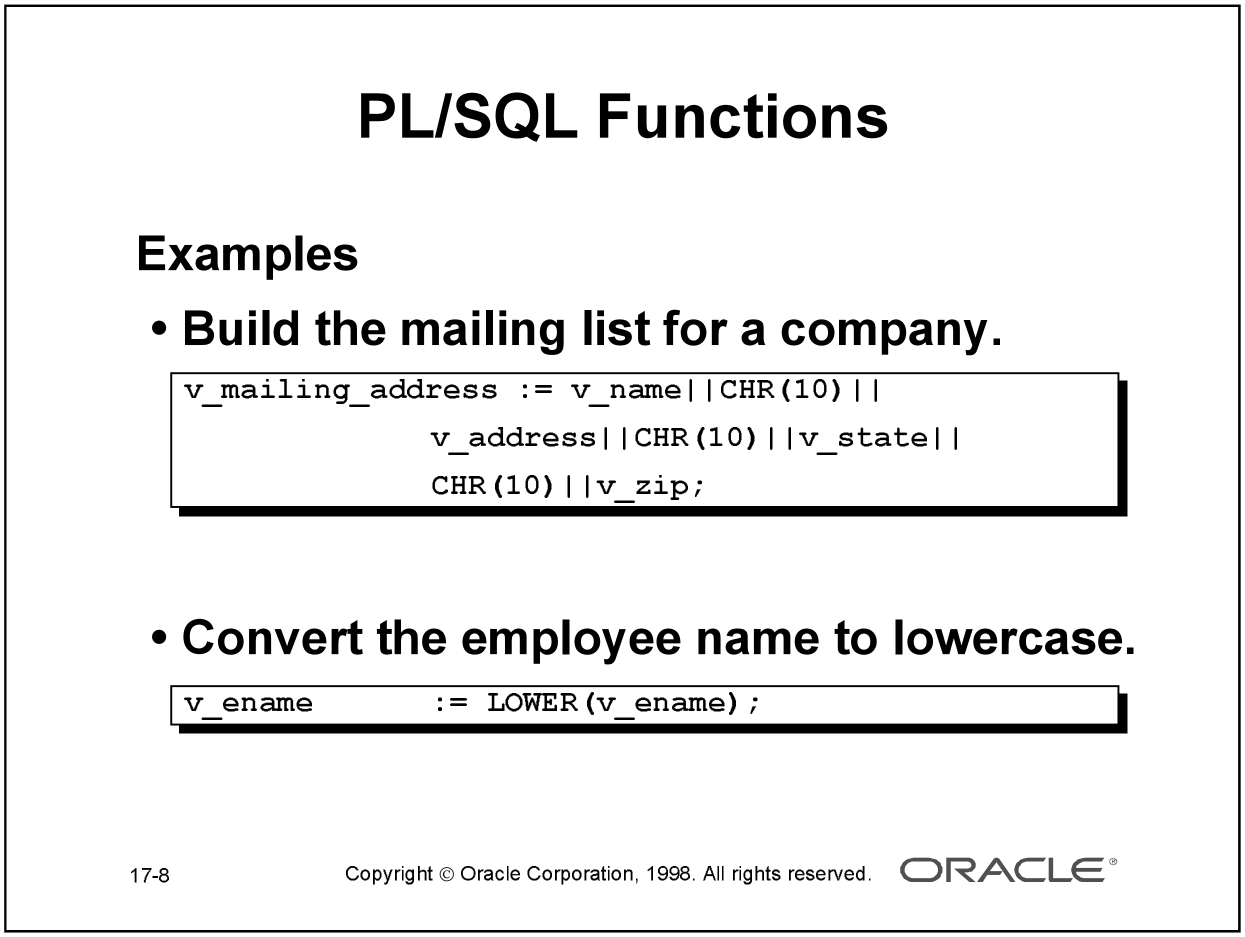
The following functions are not available in procedural statements:

- GREATEST, LEAST, and DECODE.
- Group functions: AVG, MIN, MAX, COUNT, SUM, STDDEV, and VARIANCE. Group functions apply to groups of rows in a table and are therefore are available only within SQL statements in a PL/SQL block.

Example

Compute the sum of all numbers stored in the NUMBER_TABLE PL/SQL table. *This example produces a compile error.*

```
v_total     := SUM(number_table);
```



PL/SQL Functions

Examples

- **Build the mailing list for a company.**

```
v_mailing_address := v_name||CHR(10)||
             v_address||CHR(10)||v_state||
             CHR(10)||v_zip;
```

- **Convert the employee name to lowercase.**

```
v_ename       := LOWER(v_ename);
```

17-8

PL/SQL Functions

PL/SQL provides many powerful functions to help you manipulate data. These built-in functions fall into the following categories:

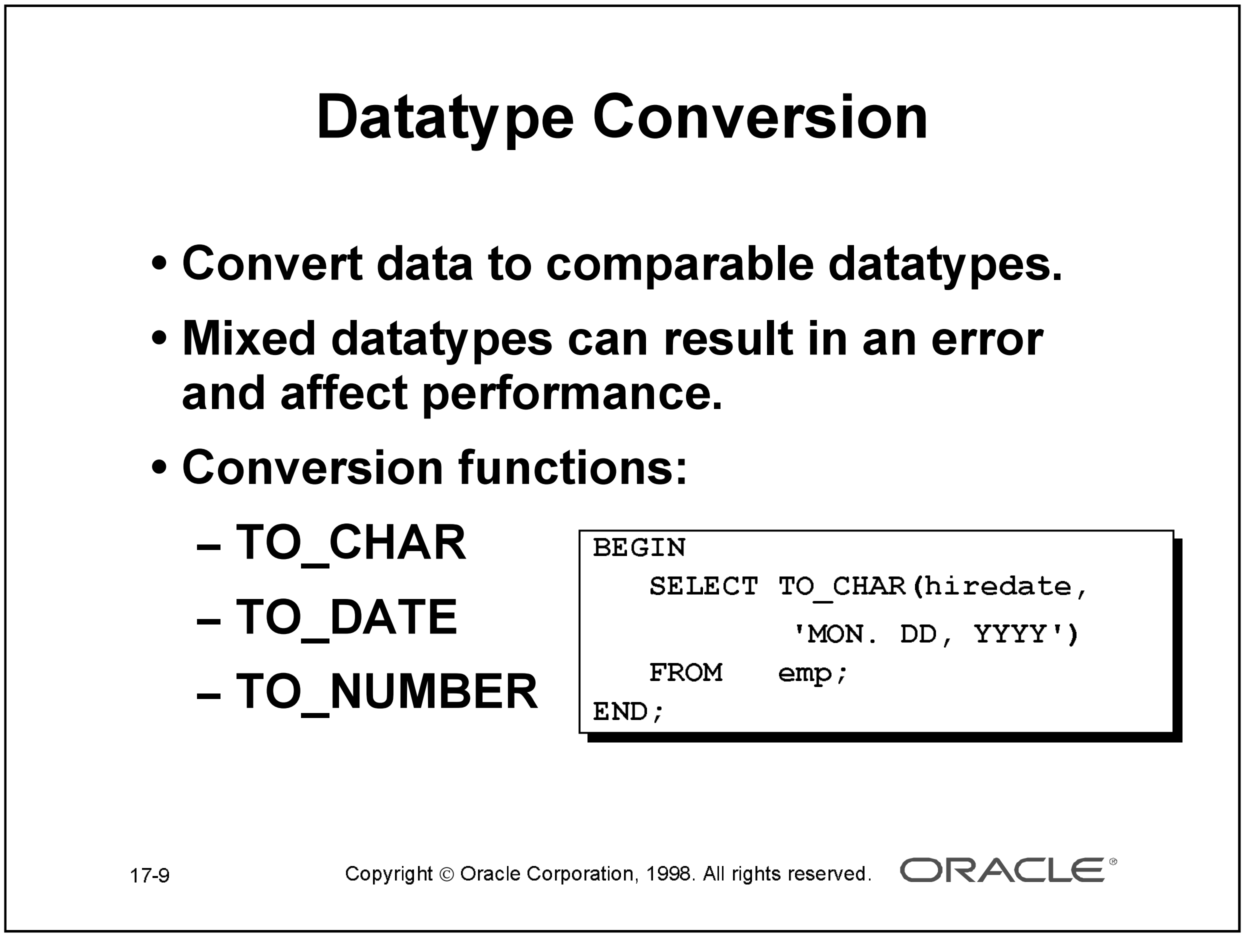
- Error-reporting
- Number
- Character
- Conversion
- Date
- Miscellaneous

The function examples in the slide are defined as follows:

- Build the mailing address for a company.
- Convert the name to lowercase.

CHR is the SQL function that converts an ASCII code to its corresponding character; 10 is the code for a line feed.

For more information, see
PL/SQL User's Guide and Reference, Release 8, "Fundamentals."



Datatype Conversion

- Convert data to comparable datatypes.
- Mixed datatypes can result in an error and affect performance.
- Conversion functions:
  - TO_CHAR
  - TO_DATE
  - TO_NUMBER

```
BEGIN
   SELECT TO_CHAR(hiredate,
                  'MON. DD, YYYY')
   FROM   emp;
END;
```

17-9

Datatype Conversion

PL/SQL attempts to convert datatypes dynamically if they are mixed within a statement. For example, if you assign a NUMBER value to a CHAR variable, then PL/SQL dynamically translates the number into a character representation, so that it can be stored in the CHAR variable. The reverse situation also applies, providing that the character expression represents a numeric value.

Providing that they are compatible, you can also assign characters to DATE variables, and vice versa.

Within an expression, you should make sure that datatypes are the same. If mixed datatypes occur in an expression, you should use the appropriate conversion function to convert the data.

Syntax

```
TO_CHAR (value, fmt)
```

```
TO_DATE (value, fmt)
```

```
TO_NUMBER (value, fmt)
```

where:	*value*	is a character string, number, or date.
	fmt	is the format model used to convert value.

Datatype Conversion

This statement produces a compile error.

```
v_comment := USER||': '||SYSDATE;
```

To correct the error, the TO_CHAR conversion function is used.

```
v_comment := USER||': '||TO_CHAR(SYSDATE);
```

17-10 ORACLE®

Datatype Conversion

The conversion examples in the slide are defined as follows:

- Store a value that is composed of the user name and today's date. *This code causes a syntax error.*
- To correct the error, convert SYSDATE to a character string with the TO_CHAR conversion function.

PL/SQL attempts conversion if possible, but the success depends on the operations being performed. It is good programming practice to explicitly perform datatype conversions, because they can favorably affect performance and remain valid even with a change in software versions.

Nested Blocks and Variable Scope

- Statements can be nested wherever an executable statement is allowed.
- A nested block becomes a statement.
- An exception section can contain nested blocks.
- The scope of an object is the region of the program that can refer to the object.

17-11

Nested Blocks

One of the advantages that PL/SQL has over SQL is the ability to nest statements. You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. Therefore you can break down the executable part of a block into smaller blocks. The exception section can also contain nested blocks.

Variable Scope

The scope of an object is the region of the program that can refer to the object. You can reference the declared variable within the executable section.

Nested Blocks and Variable Scope

An identifier is visible in the regions in which you can reference the unqualified identifier:

- A block can look up to the enclosing block.
- A block cannot look down to enclosed blocks.

17-12

Identifiers

An identifier is visible in the block in which it is declared and in all nested subblocks, procedures, and functions. If the block does not find the identifier declared locally, it looks *up* to the declarative section of the enclosing (or parent) blocks. The block never looks *down* to enclosed (or child) blocks or sideways to sibling blocks.

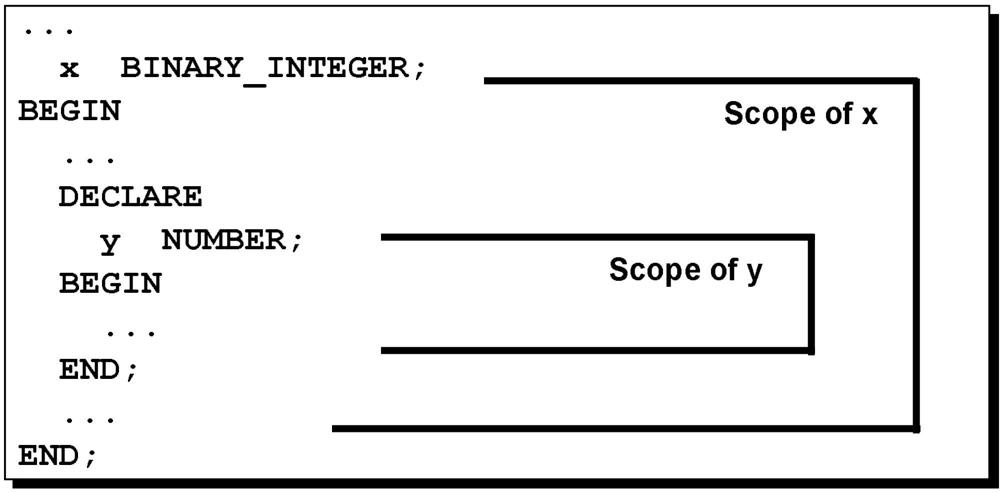
Scope applies to all declared objects, including variables, cursors, user-defined exceptions, and constants.

Note: Qualify an identifier by using the block label prefix.

For more information on block labels, see *PL/SQL User's Guide and Reference, Release 8,* "Fundamentals."

Nested Blocks and Variable Scope

Example



```
...
  x  BINARY_INTEGER;
BEGIN
  ...
  DECLARE
    y  NUMBER;
  BEGIN
    ...
  END;
  ...
END;
```

Nested Blocks and Variable Scope

In the nested block shown in the slide, the variable named *y* can reference the variable named *x*. Variable *x,* however, cannot reference variable *y.* Had the variable named *y* in the nested block been given the same name as the variable named *x* in the outer block it's value is valid only for the duration of the nested block.

Scope

The scope of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier.

Visibility

An identifier is visible only in the regions from which you can reference the identifier using an unqualified name.

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations
- Exponential operator (**)

} Same as in SQL

17-14 ORACLE®

Order of Operations

The operations within an expression are done in a particular order depending on their precedence (priority). The following table shows the default order of operations from top to bottom.

Operator	Operation
**, NOT	Exponentiation, logical negation
+, -	Identity, negation
*, /	Multiplication, division
+, -, ||	Addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparison
AND	Conjunction
OR	Inclusion

Note: It is not necessary to use parentheses with Boolean expressions, but it does make the text easier to read.

For more information on operators, see
PL/SQL User's Guide and Reference, Release 8, "Fundamentals."

Operators in PL/SQL

Examples

- **Increment the index for a loop.**

```
v_count       := v_count + 1;
```

- **Set the value of a Boolean flag.**

```
v_equal       := (v_n1 = v_n2);
```

- **Validate an employee number if it contains a value.**

```
v_valid       := (v_empno IS NOT NULL);
```

Operators in PL/SQL

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Comparisons involving nulls always yield NULL
- Applying the logical operator NOT to a null yields NULL
- In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed

Using Bind Variables

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).

Example

```
DECLARE
   v_sal      emp.sal%TYPE;
BEGIN
   SELECT     sal
   INTO       v_sal
   FROM       emp
   WHERE      empno = 7369;
  :salary     := v_sal;
END;
```

17-16 ORACLE®

Printing Bind Variables

In SQL*Plus you can display the value of the bind variable using the PRINT command.

```
SQL>  PRINT salary

SALARY
------
   800
```

Programming Guidelines

Make code maintenance easier by:

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

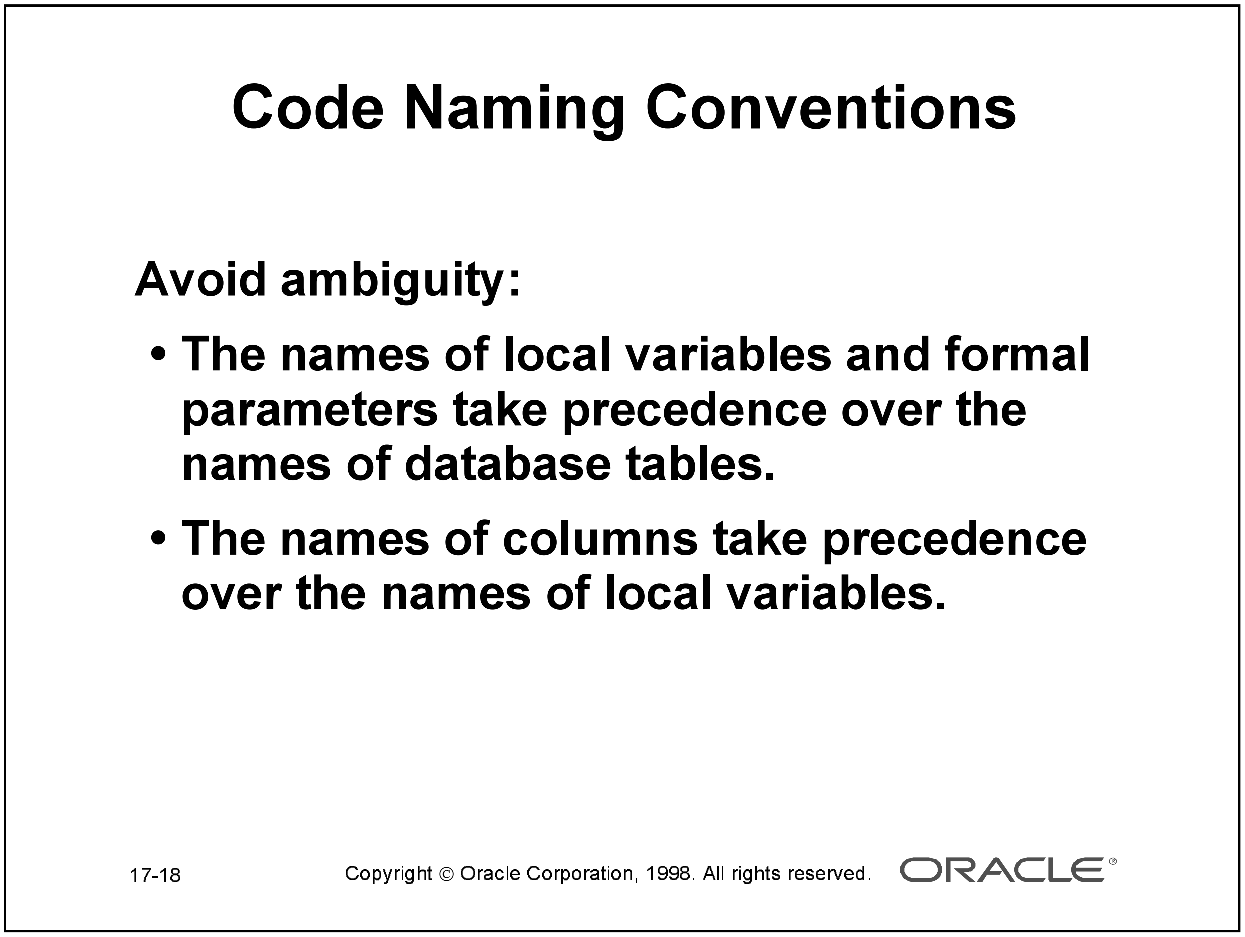
Programming Guidelines

Follow these programming guidelines to produce clear code and reduce maintenance when developing a PL/SQL block.

Code Conventions

The following table gives guidelines for writing code in uppercase or lowercase to help you to distinguish keywords from named objects.

Category	Case Convention	Examples
SQL statements	Uppercase	SELECT, INSERT
PL/SQL keywords	Uppercase	DECLARE, BEGIN, IF
Datatypes	Uppercase	VARCHAR2, BOOLEAN
Identifiers and parameters	Lowercase	v_sal, emp_cursor, g_sal, p_empno
Database tables and columns	Lowercase	emp, orderdate, deptno



Code Naming Conventions

Avoid ambiguity:

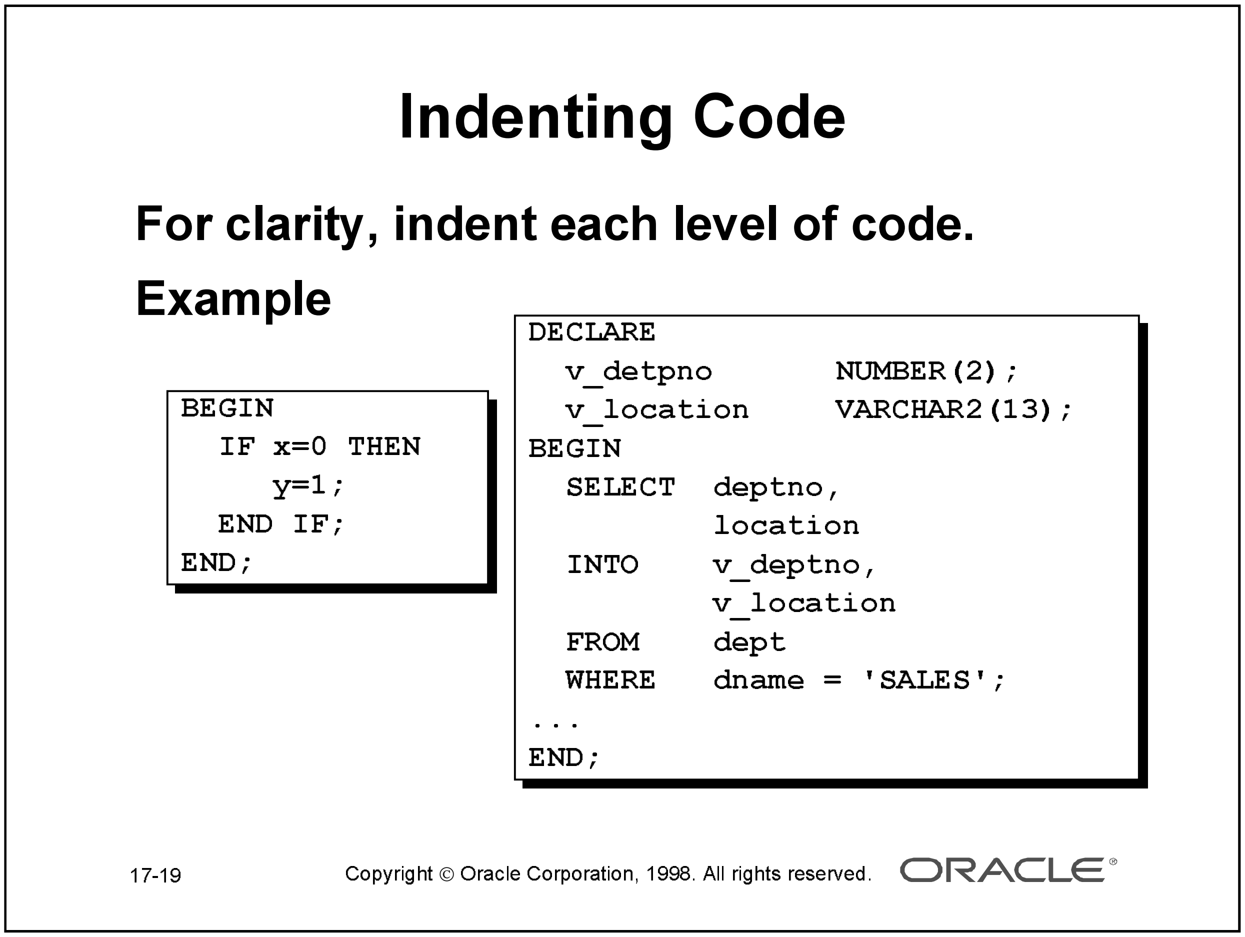
- The names of local variables and formal parameters take precedence over the names of database tables.
- The names of columns take precedence over the names of local variables.

17-18

Code Naming Conventions

The following table shows a set of prefixes and suffixes to distinguish identifiers from other identifiers, from database objects, and from other named objects.

Identifier	**Naming Convention**	**Example**
Variable	v_*name*	v_sal
Constant	c_*name*	c_company_name
Cursor	*name*_cursere	emp_cursor
Exception	e_*name*	e_too_many
Table Type	*name*_table_type	amount_table_type
Table	*name*_table	order_total_table
Record Type	*name*_record_type	emp_record_type
Record	*name*_record	customer_record
SQL*Plus substitution parameter	p_*name*	p_sal
SQL*Plus global variable	g_*name*	g_year_sal



Indenting Code

For clarity, indent each level of code.

Example

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

```
DECLARE
  v_detpno        NUMBER(2);
  v_location      VARCHAR2(13);
BEGIN
  SELECT  deptno,
          location
  INTO    v_deptno,
          v_location
  FROM    dept
  WHERE   dname = 'SALES';
...
END;
```

17-19 ORACLE®

Indenting Code

For clarity, and to enhance readability, indent each level of code. To show structure, you can divide lines using carriage returns and indent lines using spaces or tabs. Compare the following IF statements for readability:

```
IF x>y THEN max:=x;ELSE max:=y;END IF;
```

```
IF x > y THEN
   max := x;
ELSE
   max := y;
END IF;
```

Determine Variable Scope

Class Exercise

```
...
DECLARE
V_SAL          NUMBER(7,2) := 60000;
V_COMM         NUMBER(7,2) := V_SAL / .20;
V_MESSAGE      VARCHAR2(255) := ' eligible for commission';
BEGIN ...
  DECLARE
    V_SAL              NUMBER(7,2) := 50000;
    V_COMM             NUMBER(7,2) := 0;
    V_TOTAL_COMP       NUMBER(7,2) := V_SAL + V_COMM;
  BEGIN ...
    V_MESSAGE := 'CLERK not'||V_MESSAGE;
  END;
    V_MESSAGE := 'SALESMAN'||V_MESSAGE;
END;
```

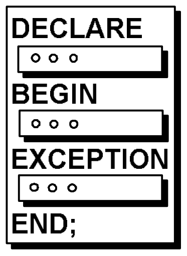
Class Exercise

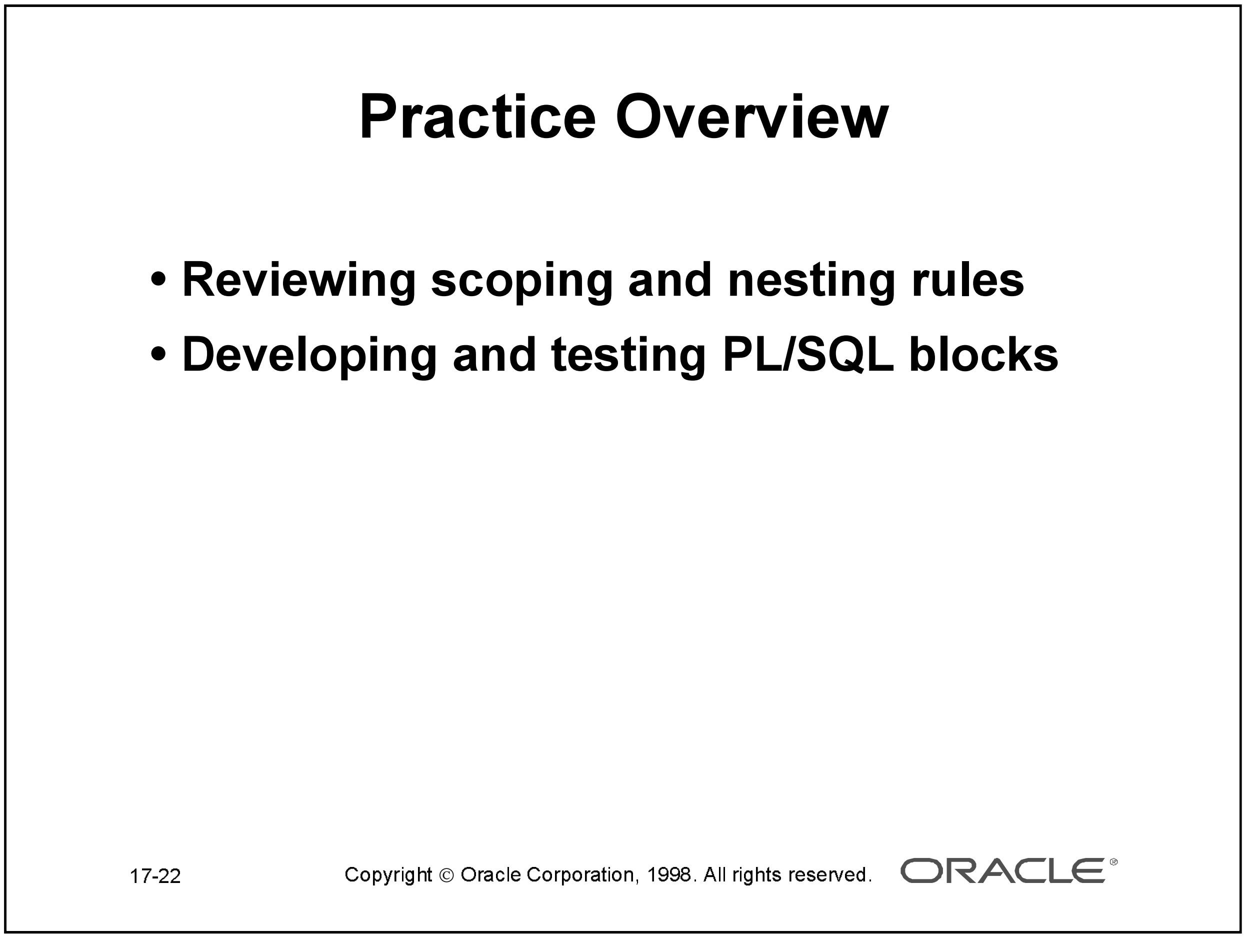
Evaluate the PL/SQL block on the slide. Determine each of the following values according to the rules of scoping:

1. The value of V_MESSAGE in the subblock.
2. The value of V_TOTAL_COMP in the main block.
3. The value of V_COMM in the subblock.
4. The value of V_COMM in the main block.
5. The value of V_MESSAGE in the main block.

Summary

- PL/SQL block structure:
  - Nesting blocks and scoping rules
- PL/SQL programming:
  - Functions
  - Datatype conversions
  - Operators
  - Bind variables
  - Conventions and guidelines





Practice Overview

- Reviewing scoping and nesting rules
- Developing and testing PL/SQL blocks

17-22

Practice Overview

This practice reinforces the basics of PL/SQL presented in the lesson, including the rules for nesting PL/SQL blocks of code as well as how to execute and test their PL/SQL code.

Paper-Based Questions

Questions 1 and 2 are paper-based questions.

Practice 17

PL/SQL Block

```
DECLARE
  v_weight  NUMBER(3) := 600;
  v_message VARCHAR2(255) := 'Product 10012';
BEGIN

            SUB-BLOCK
    DECLARE
      v_weight     NUMBER(3) := 1;
      v_message    VARCHAR2(255) := 'Product 11001';
      v_new_locn   VARCHAR2(50) := 'Europe';
    BEGIN
      v_weight := v_weight + 1;
      v_new_locn := 'Western ' || v_new_locn;
    END;

  v_weight := v_weight + 1;
  v_message := v_message || ' is in stock';
  v_new_locn := 'Western ' || v_new_locn;

END;
```

Practice 17 (continued)

1. Evaluate the PL/SQL block on the previous page and determine each of the following values according to the rules of scoping.

   a. The value of V_WEIGHT in the subblock is

   ____

   b. The value of V_NEW_LOCN in the subblock is

   ____

   c. The value of V_WEIGHT in the main block is

   ____

   d. The value of V_MESSAGE in the main block is

   ____

   e. The value of V_NEW_LOCN in the main block is

   ____

Practice 17 (continued)

Scope Example

```
DECLARE
  v_customer         VARCHAR2(50) := 'Womansport';
  v_credit_rating    VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
      v_customer NUMBER(7) := 201;
      v_name VARCHAR2(25) := 'Unisports';
    BEGIN
      v_customer     v_name   v_credit_rating
    END;

      v_customer     v_name   v_credit_rating

END;
```

Practice 17 (continued)

2. Suppose you embed a subblock within a block, as shown on the previous page. You declare two variables, V_CUSTOMER and V_CREDIT_RATING, in the main block. You also declare two variables, V_CUSTOMER and V_NAME, in the subblock. Determine the values for each of the following cases.

   a. The value of V_CUSTOMER in the subblock is

   ____

   b. The value of V_NAME in the subblock is

   ____

   c. The value of V_CREDIT_RATING in the subblock is

   ____

   d. The value of V_CUSTOMER in the main block is

   ____

   e. The value of V_NAME in the main block is

   ____

   f. The value of V_CREDIT_RATING in the main block is

   ____

3. Create and execute a PL/SQL block that accepts two numbers through SQL*Plus substitution variables. The first number should be divided by the second number and have the second number added to the result. The result should be written to a PL/SQL variable and printed to the screen.

```
Please enter the first number: 2
Please enter the second number: 4

PL/SQL procedure successfully completed.

V_RESULT
--------
    4.5
```

4. Build a PL/SQL block that computes the total compensation for one year. The annual salary and the annual bonus percentage are passed to the PL/SQL block through SQL*Plus substitution variables and the bonus needs to be converted from a whole number to a decimal (for example, 15 to .15). If the salary is null, set it to zero before computing the total compensation. Execute the PL/SQL block. Reminder: Use the NVL function to handle null values.

   Note: To test the NVL function type NULL at the prompt; pressing [Return] results in a missing expression error.

```
Please enter the salary amount: 50000
Please enter the bonus percentage: 10

PL/SQL procedure successfully completed.

G_TOTAL
-------
  55000
```

18

Interacting with the Oracle Server

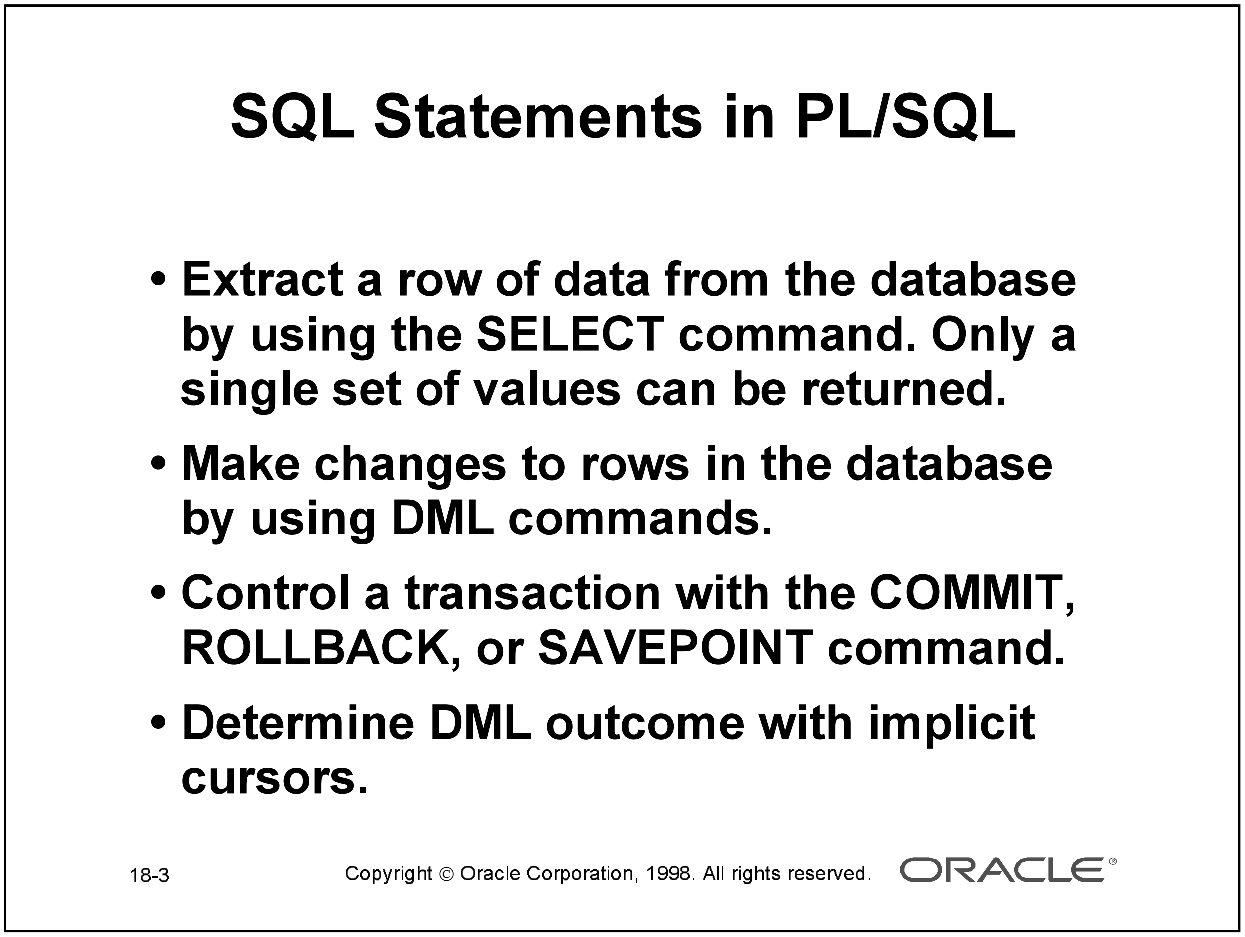
Objectives

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

- **Write a successful SELECT statement in PL/SQL**
- **Declare the datatype and size of a PL/SQL variable dynamically**
- **Write DML statements in PL/SQL**
- **Control transactions in PL/SQL**
- **Determine the outcome of SQL DML statements**

Lesson Aim

In this lesson, you will learn to embed standard SQL SELECT, INSERT, UPDATE, and DELETE statements in PL/SQL blocks. You will also learn how to control transactions and determine the outcome of SQL DML statements in PL/SQL.



SQL Statements in PL/SQL

- Extract a row of data from the database by using the SELECT command. Only a single set of values can be returned.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the COMMIT, ROLLBACK, or SAVEPOINT command.
- Determine DML outcome with implicit cursors.

18-3 ORACLE®

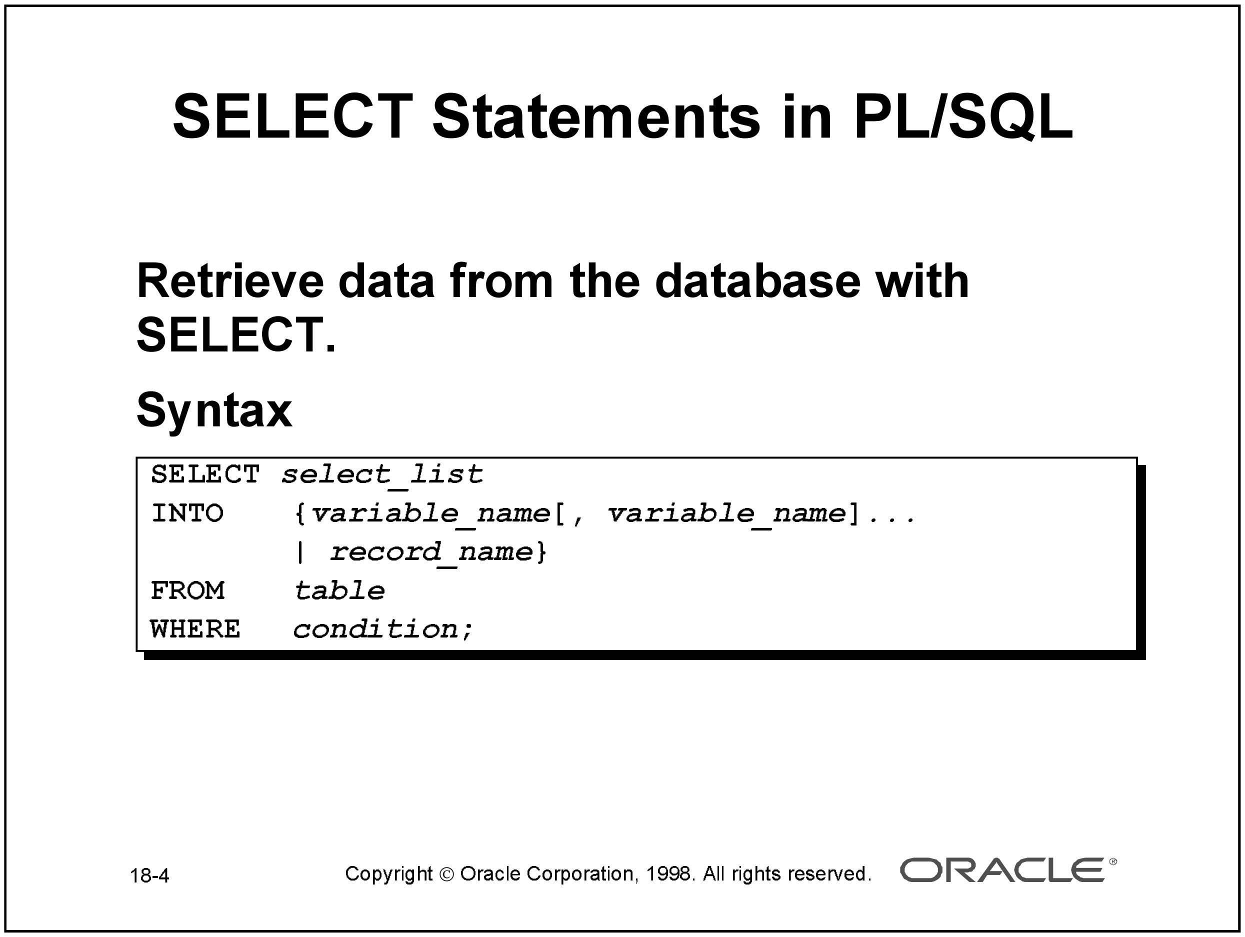
Overview

When you need to extract information from or apply changes to the database, you must use SQL. PL/SQL supports full data manipulation language and transaction control commands within SQL. You can use SELECT statements to populate variables with values queried from a row in a table. Your DML (data manipulation) commands can process multiple rows.

Comparing SQL and PL/SQL Statement Types

- A PL/SQL block is not a transaction unit. Commits, savepoints, and rollbacks are independent of blocks, but you can issue these commands within a block.
- PL/SQL does not support data definition language (DDL), such as CREATE TABLE, ALTER TABLE, or DROP TABLE.
- PL/SQL does not support data control language (DCL), such as GRANT or REVOKE.

For more information about the DBMS_SQL package, see
Oracle8 Server Application Developer's Guide, Release 8.



SELECT Statements in PL/SQL

Retrieve data from the database with SELECT.

Syntax

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

18-4 Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

Retrieving Data Using PL/SQL

Use the SELECT statement to retrieve data from the database.

In the syntax,

select_list	is a list of at least one column, and can include SQL expressions, row functions, or group functions.
variable_name	is the scalar variable to hold the retrieved value or.
record_name	is the PL/SQL RECORD to hold the retrieved values.
table	specifies the database table name.
condition	is composed of column names, expressions, constants, and comparison operators, including PL/SQL variables and constants.

Take advantage of the full range of Oracle Server syntax for the SELECT statement.

Remember that host variables must be prefixed with a colon.

SELECT Statements in PL/SQL

INTO clause is required.

Example

```
DECLARE
  v_deptno    NUMBER(2);
  v_loc       VARCHAR2(15);
BEGIN
  SELECT      deptno, loc
    INTO      v_deptno, v_loc
  FROM        dept
  WHERE       dname = 'SALES';
...
END;
```

INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables to hold the values that SQL returns from the SELECT clause. You must give one variable for each item selected, and their order must correspond to the items selected.

You use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return One and Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of Embedded SQL, for which the following rule applies: queries must return one and only one row. More than one row or no row generates an error.

⚠ PL/SQL deals with these errors by raising standard exceptions, which you can trap in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions (exception handling is covered in a subsequent lesson). You should code SELECT statements to return a single row.

Retrieving Data in PL/SQL

Retrieve the order date and the ship date for the specified order.

Example

```
DECLARE
  v_orderdate   ord.orderdate%TYPE;
  v_shipdate    ord.shipdate%TYPE;
BEGIN
  SELECT   orderdate, shipdate
    INTO   v_orderdate, v_shipdate
  FROM     ord
  WHERE    id = 157;
       ...
END;
```

18-6 ORACLE®

Guidelines

Follow these guidelines to retrieve data in PL/SQL:

- Terminate each SQL statement with a semicolon (;).
- The INTO clause is required for the SELECT statement when it is embedded in PL/SQL.
- The WHERE clause is optional and can be used to specify input variables, constants, literals, or PL/SQL expressions.
- Specify the same number of output variables in the INTO clause as database columns in the SELECT clause. Be sure that they correspond positionally and that their datatypes are compatible.

Retrieving Data in PL/SQL

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

Example

```
DECLARE
  v_sum_sal   emp.sal%TYPE;
  v_deptno     NUMBER NOT NULL := 10;
BEGIN
  SELECT     SUM(sal)  -- group function
    INTO     v_sum_sal
  FROM       emp
  WHERE      deptno = v_deptno;
END;
```

18-7

Guidelines (continued)

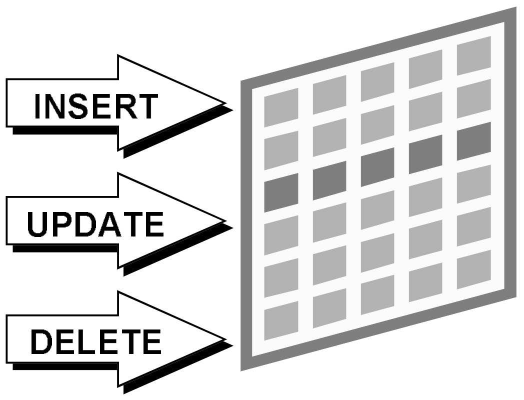
- To ensure that the datatypes of the identifiers match the datatypes of the columns use the %TYPE attribute. The datatype and number of variables in the INTO clause match those in the SELECT list.
- Use group functions, such as SUM, in a SQL statement, because group functions apply to groups of rows in a table.

Note: Group functions can not be used in PL/SQL syntax, they are used in SQL statements within a PL/SQL block.

Manipulating Data Using PL/SQL

Make changes to database tables by using DML commands:

- **INSERT**
- **UPDATE**
- **DELETE**



Manipulating Data Using PL/SQL

You manipulate data in the database by using the DML (data manipulation) commands. You can issue the DML commands INSERT, UPDATE, and DELETE without restriction in PL/SQL. By including COMMIT or ROLLBACK statements in the PL/SQL code, row locks (and table locks) are released.

- INSERT statement adds new rows of data to the table.
- UPDATE statement modifies existing rows in the table.
- DELETE statement removes unwanted rows from the table.

Inserting Data

Add new employee information to the emp table.

Example

```
DECLARE
  v_empno       emp.empno%TYPE;
BEGIN
  SELECT        empno_sequence.NEXTVAL
    INTO        v_empno
    FROM        dual;
  INSERT INTO   emp(empno, ename, job, deptno)
    VALUES(v_empno, 'HARDING', 'CLERK', 10);
END;
```

18-9 ORACLE®

Inserting Data

- Use SQL functions, such as USER and SYSDATE.
- Generate primary key values by using database sequences.
- Derive values in the PL/SQL block.
- Add column default values.

Note: There is no possibility for ambiguity with identifiers and column names in the INSERT statement. Any identifier in the INSERT clause must be a database column name.

Updating Data

Increase the salary of all employees in the emp table who are Analysts.

Example

```
DECLARE
  v_sal_increase   emp.sal%TYPE := 2000;
BEGIN
  UPDATE      emp
    SET       sal = sal + v_sal_increase
    WHERE     job = 'ANALYST';
END;
```

Updating and Deleting Data

There may be ambiguity in the SET clause of the UPDATE statement because although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable.

Remember that the WHERE clause is used to determine which rows are affected. If no rows are modified, no error occurs, unlike the SELECT statement in PL/SQL.

Note: PL/SQL variable assignments always use := and SQL column assignments always use =. Recall that if column names and identifier names are identical in the WHERE clause, the Oracle Server looks to the database first for the name.

Deleting Data

Delete rows that have belong to department 10 from the emp table.

Example

```
DECLARE
  v_deptno   emp.deptno%TYPE := 10;
BEGIN
  DELETE FROM emp
    WHERE deptno = v_deptno;
END;
```

18-11

Deleting Data

Delete a specified order.

```
DECLARE
  v_ordid  ord.ordid%TYPE := 605;
BEGIN
  DELETE FROM  item
    WHERE      ordid = v_ordid;
END;
```

Naming Conventions

- Use a naming convention to avoid ambiguity in the WHERE clause.
- Database columns and identifiers should have distinct names.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.

Avoid ambiguity in the WHERE clause by adhering to a naming convention that distinguishes database column names from PL/SQL variable names.

Example

Retrieve the date ordered and the date shipped from the ord table where the date shipped is today. This example raises an unhandled runtime exception.

```
DECLARE
  order_date ord.orderdate%TYPE;
  ship_date  ord.shipdate%TYPE;
  v_date DATE := SYSDATE;
BEGIN
  SELECT orderdate, shipdate
  INTO   order_date, ship_date
  FROM   ord
  WHERE  shipdate = v_date; -- unhandled exception:  NO_DATA_FOUND
END;
SQL> /
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 6
```

Naming Conventions

```
DECLARE
   order_date  ord.orderdate%TYPE;
   ship_date   ord.shipdate%TYPE;
   v_date DATE := SYSDATE;
BEGIN
   SELECT orderdate, shipdate
   INTO   order_date, ship_date
   FROM   ord
   WHERE  shipdate = v_date;
END;
SQL> /
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 6
```

18-13 ORACLE®

Naming Conventions

Avoid ambiguity in the WHERE clause by adhering to a naming convention that distinguishes database column names from PL/SQL variable names.

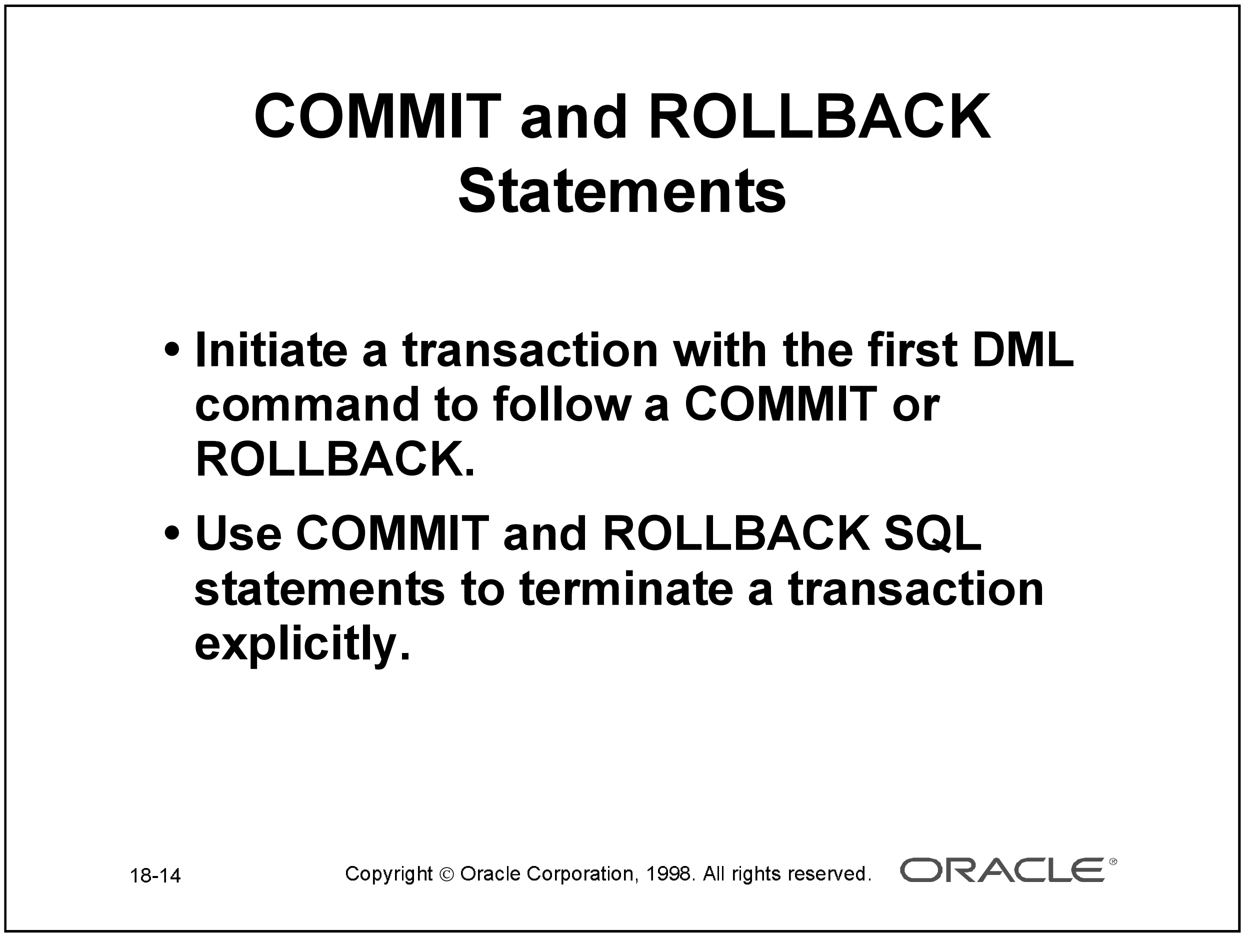
- Database columns and identifiers should have distinct names.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.

The example shown in the slide is defined as follows: retrieve the date ordered and the date shipped from the ord table where the date shipped is today. This example raises an unhandled runtime exception.

PL/SQL checks whether an identifier is a column in the database; if not, it is assumed to be a PL/SQL identifier.

Note: There is no possibility for ambiguity in the SELECT clause because any identifier in the SELECT clause must be a database column name. There is no possibility for ambiguity in the INTO clause because identifiers in the INTO clause must be PL/SQL variables. Only in the WHERE clause is there the possibility of confusion.

More information on NO_DATA_FOUND and other exceptions is covered in a subsequent lesson.



COMMIT and ROLLBACK Statements

- Initiate a transaction with the first DML command to follow a COMMIT or ROLLBACK.
- Use COMMIT and ROLLBACK SQL statements to terminate a transaction explicitly.

18-14

Controlling Transactions

You control the logic of transactions with COMMIT and ROLLBACK SQL statements, rendering some groups of database changes permanent while discarding others. As with Oracle Server, DML transactions start at the first command to follow a COMMIT or ROLLBACK and end on the next successful COMMIT or ROLLBACK. These actions may occur within a PL/SQL block or as a result of events in the host environment (for example, ending a SQL*Plus session automatically commits the pending transaction).

COMMIT Statement

COMMIT ends the current transaction by making all pending changes to the database permanent.

Syntax

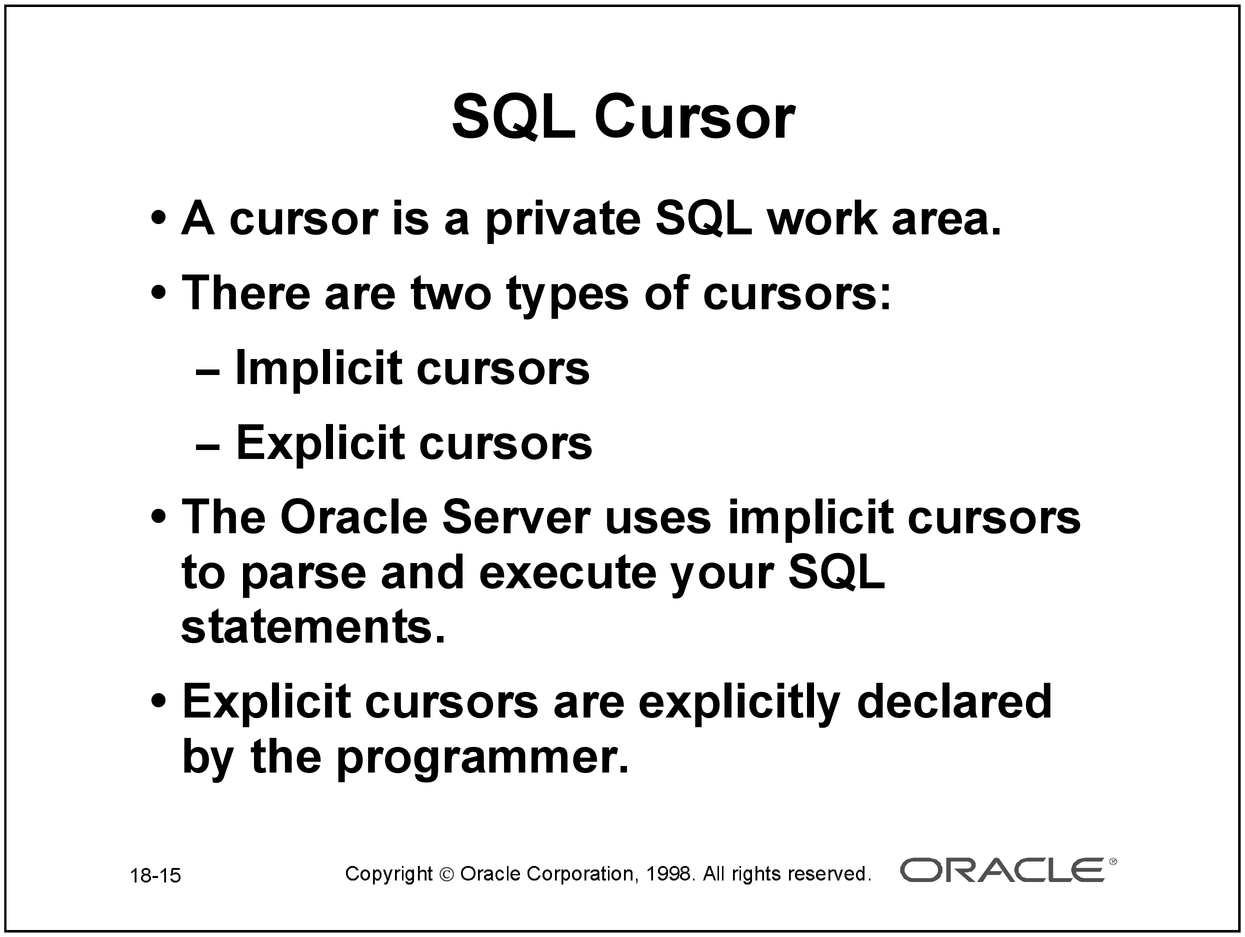
```
COMMIT [WORK];
```

```
ROLLBACK [WORK];
```

where:	WORK	is for compliance with ANSI standards.

Note: The transaction control commands are all valid within PL/SQL, although the host environment may place some restriction on their use.

You can also include explicit locking commands (such as LOCK TABLE and SELECT ... FOR UPDATE) in a block (a subsequent lesson will cover more information on the FOR_UPDATE command). They stay in effect until the end of the transaction. Also, one PL/SQL block does not necessarily imply one transaction.



SQL Cursor

- A cursor is a private SQL work area.
- There are two types of cursors:
  - Implicit cursors
  - Explicit cursors
- The Oracle Server uses implicit cursors to parse and execute your SQL statements.
- Explicit cursors are explicitly declared by the programmer.

18-15

SQL Cursor

Whenever you issue a SQL statement, the Oracle Server opens an area of memory in which the command is parsed and executed. This area is called a *cursor*.

When the executable part of a block issues a SQL statement, PL/SQL creates an implicit cursor, which has the SQL identifier. PL/SQL manages this cursor automatically. An explicit cursor is explicitly declared and named by the programmer. There are four attributes available in PL/SQL that can be applied to cursors.

Note: More information about explicit cursors is covered in a subsequent lesson.

For more information, see
PL/SQL User's Guide and Reference, Release 8, "Interaction with Oracle."

SQL Cursor Attributes

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

SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value).
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows.
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows.
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.

18-16 ORACLE®

SQL Cursor Attributes

SQL cursor attributes allow you to evaluate what happened when the implicit cursor was last used. You use these attributes in PL/SQL statements such as functions. You cannot use them in SQL statements.

You can use the attributes, SQL%ROWCOUNT, SQL%FOUND, SQL%NOTFOUND, and SQL%ISOPEN in the exception section of a block to gather information about the execution of a data manipulation statement. PL/SQL does not consider a DML statement that affects no rows to have failed, unlike the SELECT statement, which returns an exception.

SQL Cursor Attributes

Delete rows that have the specified order number from the ITEM table. Print the number of rows deleted.

Example

```
VARIABLE rows_deleted
DECLARE
  v_ordid  NUMBER := 605;
BEGIN
  DELETE FROM item
  WHERE  ordid = v_ordid;
         rows_deleted := SQL%ROWCOUNT
             ||' rows deleted.');
END;
PRINT rows_deleted
```

Summary

- Embed SQL in the PL/SQL block:
  - SELECT, INSERT, UPDATE, DELETE.
- Embed transaction control statements in a PL/SQL block:
  - COMMIT, ROLLBACK, SAVEPOINT.

Summary

- There are two cursor types: implicit and explicit.
- Implicit cursor attributes verify the outcome of DML statements:
  - SQL%ROWCOUNT
  - SQL%FOUND
  - SQL%NOTFOUND
  - SQL%ISOPEN
- Explicit cursors are defined by the user.

Practice Overview

- Creating a PL/SQL block to select data from a table
- Creating a PL/SQL block to insert data into a table
- Creating a PL/SQL block to update data in a table
- Creating a PL/SQL block to delete a record from a table

18-20 ORACLE®

Practice Overview

In this practice, you create procedures to select, input, update, and delete information in a table, using basic SQL query and DML statements within a PL/SQL block.

Practice 18

1. Create a PL/SQL block that selects the maximum department number in the DEPT table and store it in a SQL*Plus variable. Print the results to the screen. Save your PL/SQL block to a file named *p18q1.sql.*

```
G_MAX_DEPTNO
------------
          40
```

2. Create a PL/SQL block that inserts a new department into the DEPT table. Save your PL/SQL block to a file named *p18q2.sql.*

   a. Use the department number retrieved from exercise 1 and add 10 to that number as the input department number for the new department.

   b. Use a parameter for the department name.

   c. Leave the location null for now.

   d. Execute the PL/SQL block.

```
Please enter the department number: 50
Please enter the department name: EDUCATION

PL/SQL procedure successfully completed.
```

   e. Display the new department that you created.

```
  DEPTNO DNAME      LOC
  ------ ---------- -----
      50 EDUCATION
```

3. Create a PL/SQL block that updates the location for an existing department. Save your PL/SQL block to a file named *p18q3.sql.*

   a. Use a parameter for the department number.

   b. Use a parameter for the department location.

   c. Test the PL/SQL block.

```
Please enter the department number: 50
Please enter the department location: HOUSTON

PL/SQL procedure successfully completed.
```

d. Display the department number, department name, and location for the updated department.

```
DEPTNO DNAME          LOC
------ --------- ---------------
    50 EDUCATION HOUSTON
```

e. Display the department that you updated.

4. Create a PL/SQL block that deletes the department created in exercise 2. Save your PL/SQL block to a file named *p18q4.sql.*

a. Use a parameter for the department number.

b. Print to the screen the number of rows affected.

c. Test the PL/SQL block.

```
Please enter the department number: 50
PL/SQL procedure successfully completed.

G_RESULT
------------------------------------------------------------
1 row(s) deleted.
```

d. What happens if you enter a department number that does not exist?

```
Please enter the department number: 99
PL/SQL procedure successfully completed.


G_RESULT
------------------------------------------------------------
0 row(s) deleted.
```

e. Confirm that the department has been deleted.

```
no rows selected
```

19

Writing Control Structures

Objectives

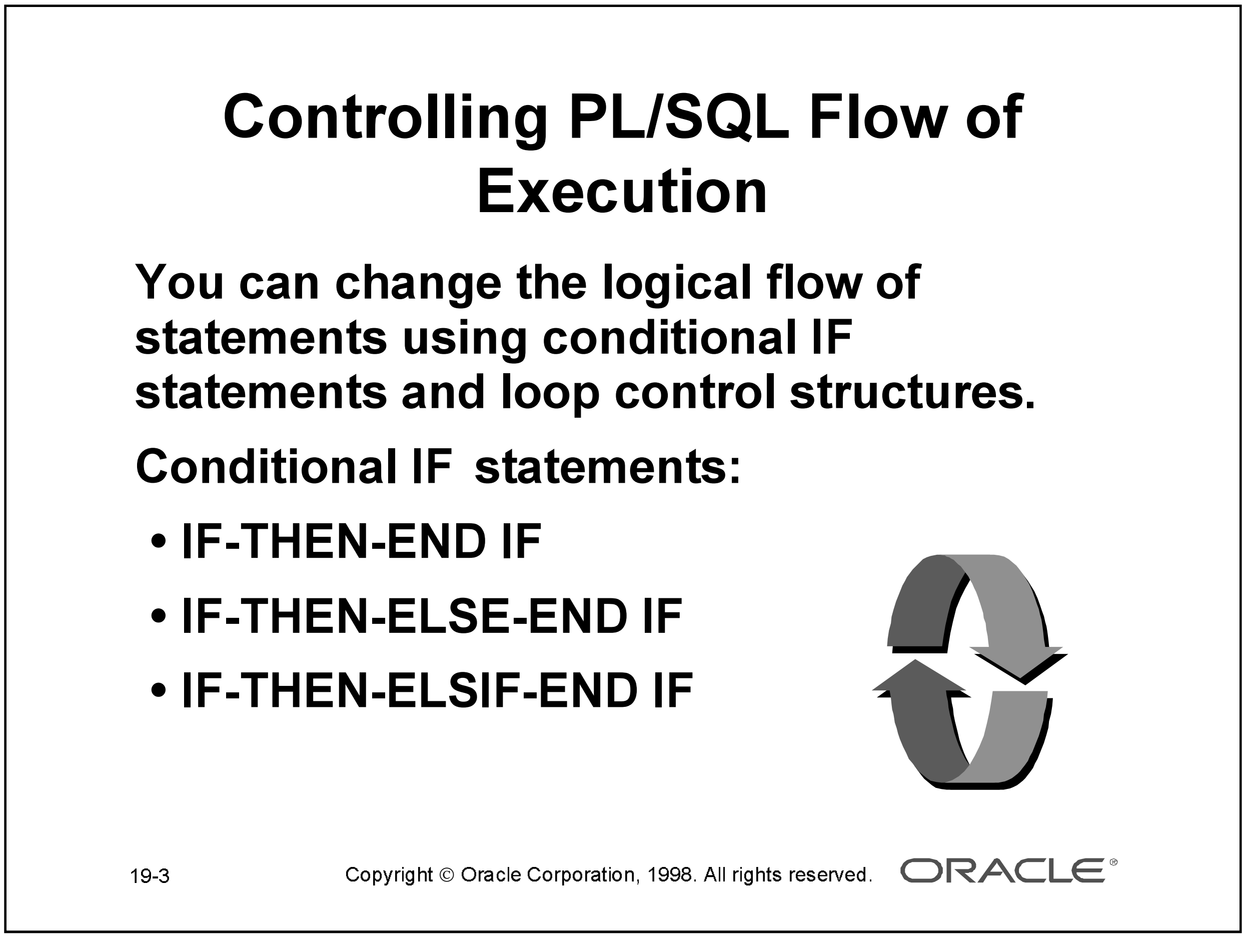
After completing this lesson, you should be able to do the following:

- Identify the uses and types of control structures
- Construct an IF statement
- Construct and identify different loop statements
- Use logic tables
- Control block flow using nested loops and labels

19-2 ORACLE®

Lesson Aim

In this lesson, you will learn about conditional control within the PL/SQL block by using IF statements and loops.



Controlling PL/SQL Flow of Execution

You can change the logical flow of statements using conditional IF statements and loop control structures.

Conditional IF statements:

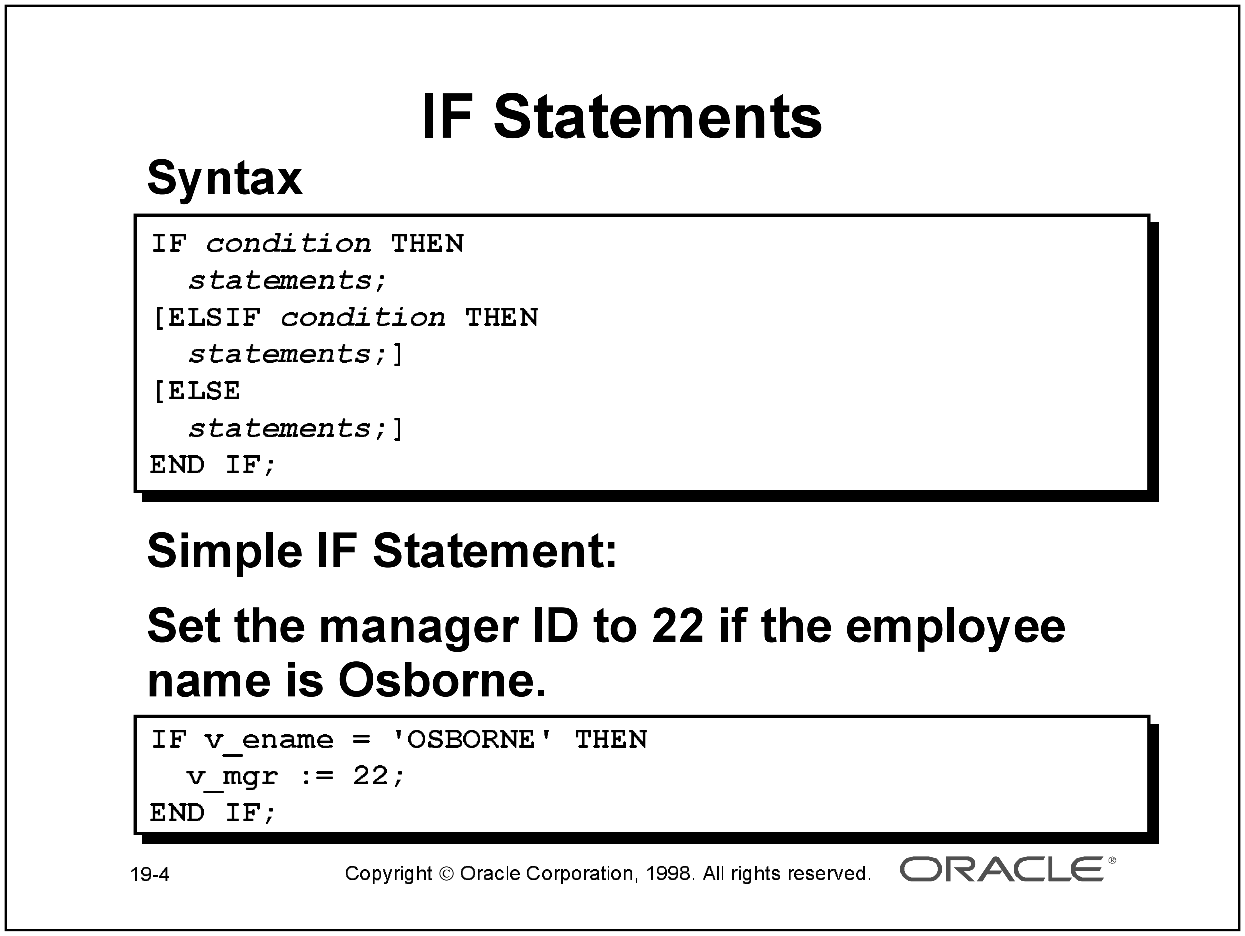
- **IF-THEN-END IF**
- **IF-THEN-ELSE-END IF**
- **IF-THEN-ELSIF-END IF**

19-3 Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

You can change the logical flow of statements within the PL/SQL block with a number of *control structures*. This lesson addresses two types of PL/SQL control structures; conditional constructs with the IF statement and and LOOP control structures (covered later in this lesson).

There are three forms of IF statements:

- IF-THEN-END IF
- IF-THEN-ELSE-END IF
- IF-THEN-ELSIF-END IF



IF Statements

Syntax

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

Simple IF Statement:

Set the manager ID to 22 if the employee name is Osborne.

```
IF v_ename = 'OSBORNE' THEN
  v_mgr := 22;
END IF;
```

19-4 ORACLE®

IF Statements

The structure of the PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It allows PL/SQL to perform actions selectively based on conditions.

In the syntax,

condition	is a Boolean variable or expression (TRUE, FALSE, or NULL). It is associated with a sequence of statements, which is executed only if the expression yields TRUE.
THEN	is a clause that associates the Boolean expression that precedes it with the sequence of statements that follows it.
statements	can be one or more PL/SQL or SQL statements. They may include further IF statements containing several nested IFs, ELSEs, and ELSIFs.
ELSIF	is a keyword that introduces a Boolean expression. If the first condition yields FALSE or NULL then the ELSIF keyword introduces additional conditions.
ELSE	is a keyword that if control reaches it, the sequence of statements that follows it is executed.

Simple IF Statements

Set the job title to Salesman, the department number to 35, and the commission to 20%of the current salary if the last name is Miller.

Example

```
. . .
IF v_ename = 'MILLER' THEN
  v_job := 'SALESMAN';
  v_deptno := 35;
  v_new_comm := sal * 0.20;
END IF;
. . .
```

19-5

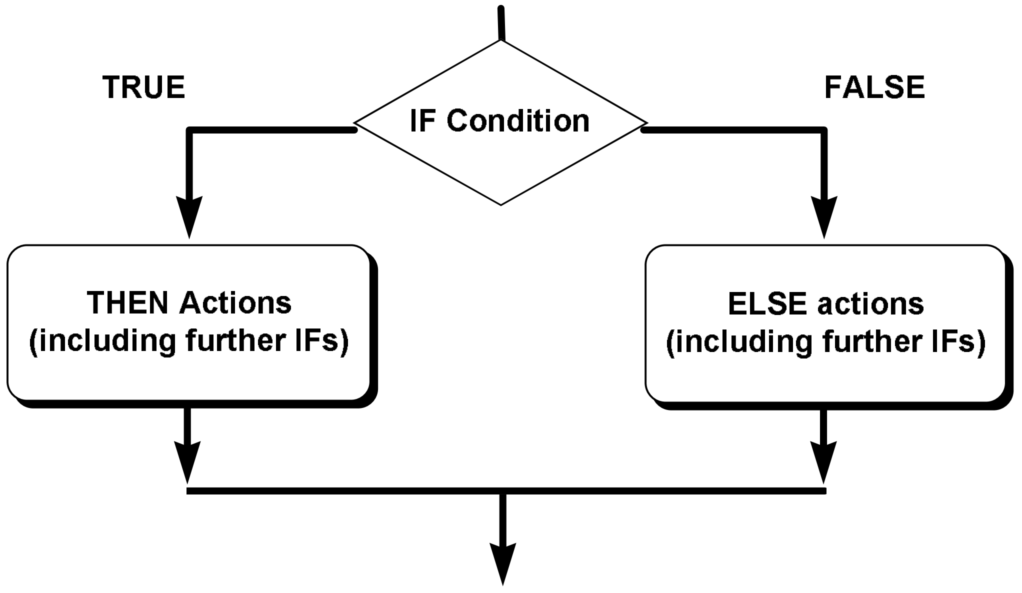
Simple IF Statements

In the example in the slide, PL/SQL performs these two actions (setting the *v_job*, *v_deptno*, and *v_new_comm* variables) only if the condition is TRUE. If the condition is FALSE or NULL, PL/SQL ignores them. In either case, control resumes at the next statement in the program following END IF.

Guidelines

- You can perform actions selectively based on conditions being met.
- When writing code, remember the spelling of the keywords:
  - ELSIF is one word.
  - END IF is two words.
- If the controlling Boolean condition is TRUE, the associated sequence of statements is executed; if the controlling Boolean condition is FALSE or NULL, the associated sequence of statements is passed over. Any number of ELSIF clauses is permitted.
- There can be at most one ELSE clause.
- Indent the conditionally executed statements for clarity.

IF-THEN-ELSE Statement Execution Flow


TRUE
IF Condition
FALSE
THEN Actions (including further IFs)
ELSE actions (including further IFs)

If the condition is FALSE or NULL, you can use the ELSE clause to carry out other actions. As with the simple IF statement, control resumes in the program from the END IF. Example:

```
IF condition1 THEN
  statement1;
ELSE
  statement2;
END IF;
```

Nested IF Statements

Either set of actions of the result of the first IF statement can include further IF statements before specific actions are performed. The THEN and ELSE clauses can include IF statements. Each nested IF statement must be terminated with a corresponding END IF.

```
IF condition1 THEN
  statement1;
ELSE
  IF condition2 THEN
    statement2;
  END IF;
END IF;
```

IF-THEN-ELSE Statements

Set a flag for orders where there are fewer than 5 days between order date and ship date.

Example

```
...
IF v_shipdate - v_orderdate < 5 THEN
  v_ship_flag := 'Acceptable';
ELSE
  v_ship_flag := 'Unacceptable';
END IF;
...
```

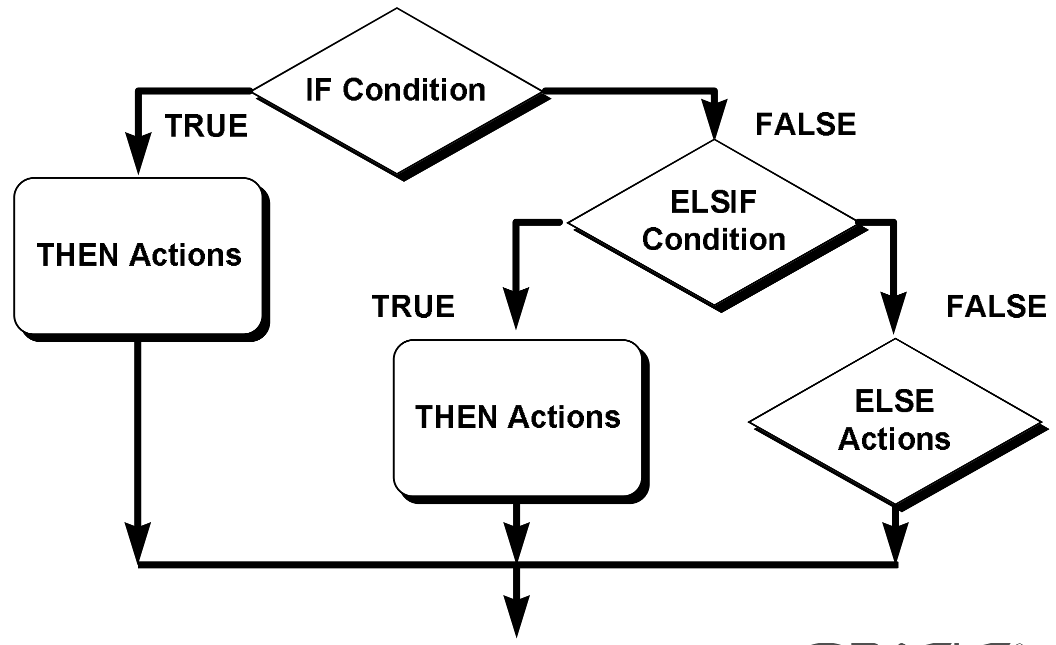
19-7

Example

Set the job to Manager if the employee name is King. If the employee name is other than King, set the job to Clerk.

```
IF v_ename = 'KING' THEN
  v_job := 'MANAGER';
ELSE
  v_job := 'CLERK';
END IF;
```

IF-THEN-ELSIF Statement Execution Flow



19-8

IF-THEN-ELSIF Statement Execution Flow

In the input screen of an Oracle Forms application, enter the department number of the new employee to determine their bonus.

```
...
IF :dept.deptno = 10 THEN
  v_comm := 5000;
ELSIF :dept.deptno = 20 THEN
  v_comm := 7500;
ELSE
  v_comm := 2000;
END IF;
...
```

In the example, the variable v_comm is used to populate a screen field with the employees bonus amount and :dept.deptno represents the value entered into the screen field.

IF-THEN-ELSIF Statements

For a given value entered, return a calculated value.

Example

```
. . .
IF v_start > 100 THEN
  v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
  v_start := .5 * v_start;
ELSE
  v_start := .1 * v_start;
END IF;
. . .
```

IF-THEN-ELSIF Statements

When possible, use the ELSIF clause instead of nesting IF statements. The code is easier to read and understand, and the logic is clearly identified. If the action in the ELSE clause consists purely of another IF statement, it is more convenient to use the ELSIF clause. This makes the code clearer by removing the need for nested END IFs at the end of each further set of conditions and actions.

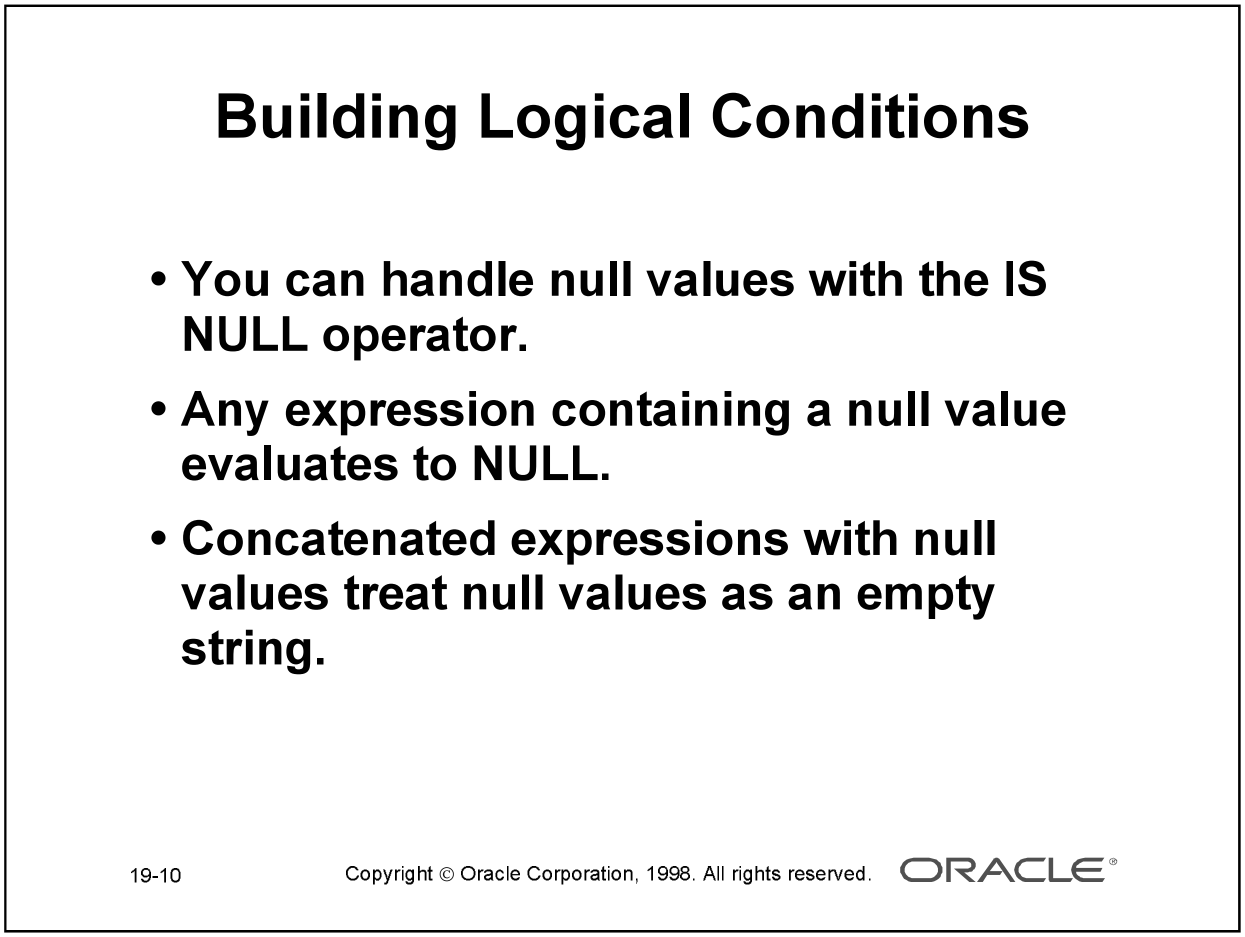
Example

```
IF condition1 THEN
  statement1;
ELSIF condition2 THEN
  statement2;
ELSIF condition3 THEN
  statement3;
END IF;
```

The example IF-THEN-ELSIF statement above is further defined as follows:

For a given value entered, return a calculated value. If the entered value is over 100, then the calculated value is two times the entered value. If the entered value is between 50 and 100, then the calculated value is 50% of the starting value. If the entered value is less than 50, then the calculated value is 10% of the starting value.

Note: Any arithmetic expression containing null values evaluates to null.



Building Logical Conditions

- You can handle null values with the IS NULL operator.
- Any expression containing a null value evaluates to NULL.
- Concatenated expressions with null values treat null values as an empty string.

19-10

Building Logical Conditions

You can build a simple Boolean condition by combining number, character, or date expressions with a comparison operator. In general, handle null values with the IS NULL operator.

Null in Expressions and Comparisons

- The IS NULL condition evaluates to TRUE only if the variable it is checking is NULL.
- Any expression containing a null value evaluates to NULL, with the exception of a concatenated expression, which treats the null value as an empty string.

Examples

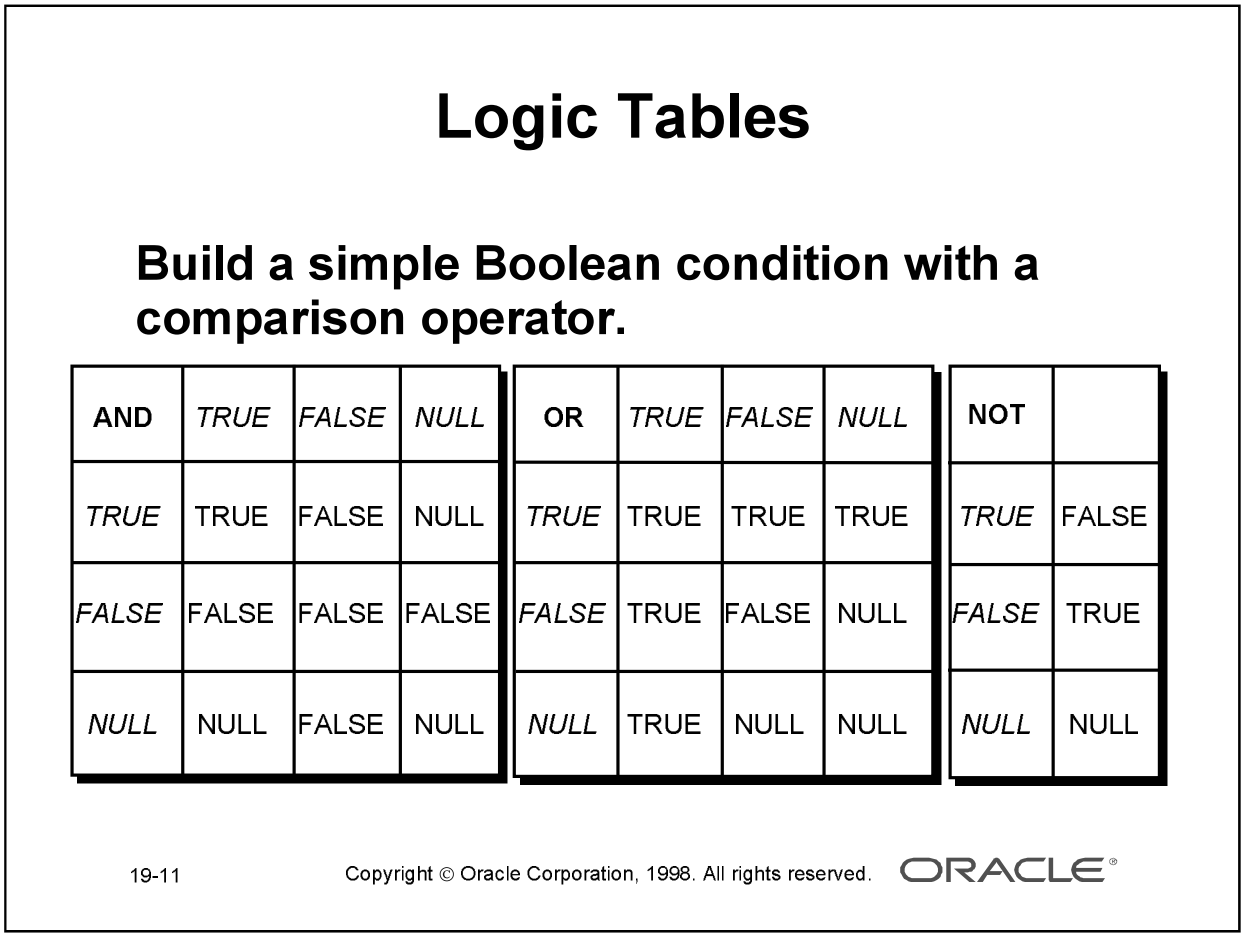
```
v_sal > 1000
```

```
v_sal * 1.1
```

v_sal evaluates to NULL if *v_sal* is NULL in both of the above examples.

In the next example the string does not evaluate to NULL if v_string is NULL.

```
'PL'||v_string||'SQL'
```



Logic Tables

Build a simple Boolean condition with a comparison operator.

AND	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

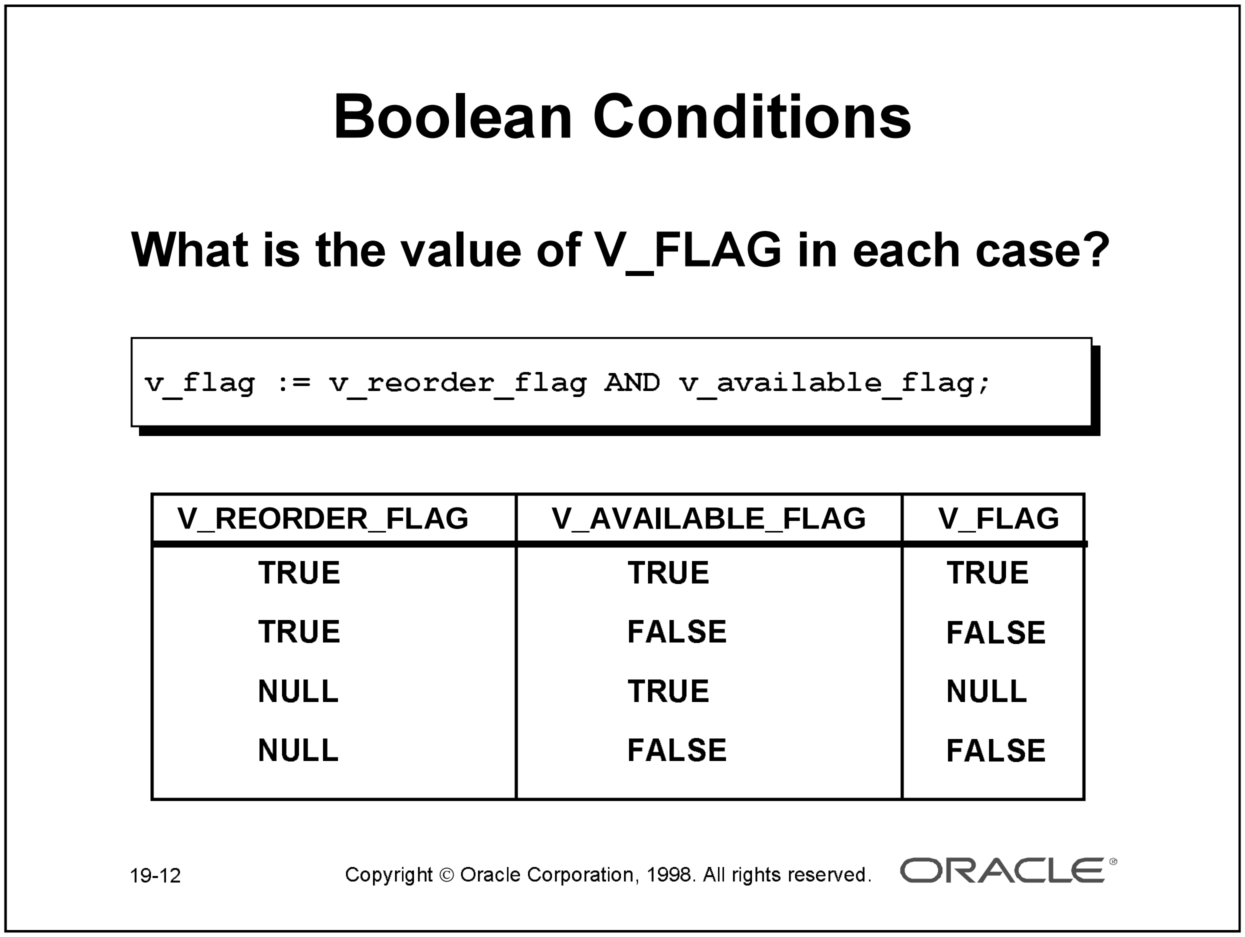
NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

19-11

Boolean Conditions with Logical Operators

You can build a complex Boolean condition by combining simple Boolean conditions with the logical operators AND, OR, and NOT. In the logic tables shown in the slide, FALSE takes precedence in an AND condition and TRUE takes precedence in an OR condition. AND returns TRUE only if both of its operands are TRUE. OR returns FALSE only if both of its operands are FALSE. NULL AND TRUE always evaluate to NULL because it is not known if the second operand evaluates to TRUE or not.

Note: The negation of NULL (NOT NULL) results in a null value because null values are indeterminate.



Boolean Conditions

What is the value of V_FLAG in each case?

```
v_flag := v_reorder_flag AND v_available_flag;
```

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
NULL	TRUE	NULL
NULL	FALSE	FALSE

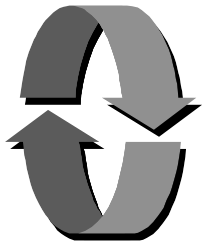
19-12

Building Logical Conditions

The AND logic table can help you evaluate the possibilities for the Boolean condition in the above slide.

Iterative Control: LOOP Statements

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



Iterative Control: LOOP Statements

PL/SQL provides a number of facilities to structure loops to repeat a statement or sequence of statements multiple times.

Looping constructs are the second type of control structure:

- Basic loop to provide repetitive actions without overall conditions
- FOR loops to provide iterative control of actions based on a count
- WHILE loops to provide iterative control of actions based on a condition
- EXIT statement to terminate loops

For more information, see
PL/SQL User's Guide and Reference, Release 8, "Control Structures."

Note: Another type of FOR LOOP, cursor FOR LOOP is discussed in a subsequent lesson.

.

Basic Loop

Syntax

```
LOOP                               -- delimiter
  statement1;                      -- statements
  . . .
  EXIT [WHEN condition];           -- EXIT statement
END LOOP;                          -- delimiter
```

```
where:   condition         is a Boolean variable or
                           expression (TRUE, FALSE,
                           or NULL);
```

Basic Loop

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP. Each time the flow of execution reaches the END LOOP statement, control is returned to the corresponding LOOP statement above it. A basic loop allows execution of its statement at least once, even if the condition is already met upon entering the loop.

- Iterate through your statements with a basic loop.
- Without the EXIT statement, the loop would be infinite.

The EXIT Statement

You can terminate a loop using the EXIT statement. Control passes to the next statement after the END LOOP statement. You can issue EXIT either as an action within an IF statement, or as a standalone statement within the loop. The EXIT statement must be placed inside a loop. In the latter case, you can attach a WHEN clause to allow conditional termination of the loop. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition yields TRUE, the loop completes and control passes to the next statement after the loop. A basic loop can contain multiple EXIT statements.

Basic Loop

Example

```
DECLARE
  v_ordid     item.ordid%TYPE := 101;
  v_counter   NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

19-15 ORACLE®

Basic Loop

The basic loop example shown in the slide is defined as follows: insert the first 10 new line items for order number 101.

Note: A basic loop allows execution of its statements at least once, even if the condition has been met upon entering the loop.

FOR Loop

Syntax

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

- **Use a FOR loop to shortcut the test for the number of iterations.**
- **Do not declare the index; it is declared implicitly.**

19-16 ORACLE®

FOR loops have the same general structure as the Basic Loop. In addition, they have a control statement at the front of the LOOP keyword to determine the number of iterations that PL/SQL performs.

In the syntax,

counter	is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached.
REVERSE	causes the index to decrement with each iteration from the upper bound to the lower bound. Note that the low value (lower bound) is still referenced first.
lower_bound	specifies the lower bound for the range of index values.
upper_bound	specifies the upper bound for the range of index values.

Do not declare the counter; it is declared implicitly as an integer.

Note: The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower bound and upper bound of the loop range can be literals, variables, or expressions, but must evaluate to integers. If the lower bound of the loop range evaluates to a larger integer than the upper bound, the sequence of statements will not be executed.

For example, statement1 is executed only once.

```
FOR i IN 3..3 LOOP statement1; END LOOP;
```

FOR Loop

Guidelines

- Reference the counter within the loop only; it is undefined outside the loop.
- Use an expression to reference the existing value of a counter.
- Do *not* reference the counter as the target of an assignment.

19-17 ORACLE®

FOR Loop

Note: The lower and upper bounds of a LOOP statement do not need to be numeric literals. They can be expressions that convert to numeric values.

Example

```
DECLARE
  v_lower   NUMBER := 1;
  v_upper   NUMBER := 100;
BEGIN
  FOR i IN v_lower..v_upper LOOP
  ...
  END LOOP;
END;
```

FOR Loop

Insert the first 10 new line items for order number 101.

Example

```
DECLARE
  v_ordid     item.ordid%TYPE := 101;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, i);
  END LOOP;
END;
```

WHILE Loop

Syntax

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

Condition is evaluated at the beginning of each iteration.

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

WHILE Loop

You can use the WHILE loop to repeat a sequence of statements until the controlling condition is no longer TRUE. The condition is evaluated at the start of each iteration. The loop terminates when the condition is FALSE. If the condition is FALSE at the start of the loop, then no further iterations are performed.

In the syntax,

condition	is a Boolean variable or expression (TRUE, FALSE, or NULL).
statement	can be one or more PL/SQL or SQL statements.

If the variables involved in the conditions do not change during the body of the loop, then the condition remains TRUE and the loop does not terminate.

Note: If the condition yields NULL, the loop is bypassed and control passes to the next statement.

WHILE Loop

Example

```
ACCEPT p_price PROMPT 'Enter the price of the item: '
ACCEPT p_itemtot PROMPT 'Enter the maximum total for
                         purchase of item: '
DECLARE
...
v_qty                NUMBER(8) := 1;
v_running_total      NUMBER(7,2) := 0;
BEGIN
  ...
  WHILE v_running_total < &p_itemtot LOOP
    ...
  v_qty := v_qty + 1;
  v_running_total := v_qty * &p_price;
  END LOOP;
...
```

19-20

WHILE Loop

In the example in the slide, the quantity increases with each iteration of the loop until the quantity is no longer less than the maximum price allowed for spending on the item.

Nested Loops and Labels

- Nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the EXIT statement referencing the label.

Nested Loops and Labels

You can nest loops to multiple levels. You can nest FOR loops within WHILE loops and WHILE loops within FOR loops. Normally the termination of a nested loop does not terminate the enclosing loop unless an exception was raised. However, you can label loops and exit the outer loop with the EXIT statement.

Label names follow the same rules as other identifiers. A label is placed before a statement, either on the same line or a separate line. Label loops by placing the label before the word LOOP within label delimiters (<<*label*>>).

If the loop is labeled, the label name can optionally be included after the END LOOP statement.

Nested Loops and Labels

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
  EXIT WHEN v_counter>10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT Outer_loop WHEN total_done = 'YES';
      -- Leave both loops
      EXIT WHEN inner_done = 'YES';
      -- Leave inner loop only
      ...
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
```

Summary

Change the logical flow of statements by using control structures.

- Conditional (IF statement)
- Loops
  - Basic loop
  - FOR loop
  - WHILE loop
  - EXIT statement

Practice Overview

- Performing conditional actions using the IF statement
- Performing iterative steps using the loop structure

Practice Overview

In this practice, you create PL/SQL blocks that incorporate loops and conditional control structures.

Practice 19

1. Run the script LABS*lab19_1.sql* to create the MESSAGES table. Write a PL/SQL block to insert numbers into the MESSAGES table.
   a. Insert the numbers 1 to 10 excluding 6 and 8.
   b. Commit before the end of the block.
   c. Select from the MESSAGES table to verify that your PL/SQL block worked.

```
RESULTS
---------
        1
        2
        3
        4
        5
        7
        9
       10
```

2. Create a PL/SQL block that computes the commission amount for a given employee based on the employee's salary.
   a. Run the script LABS*lab19_2.sql* to insert a new employee into the EMP table.
      Note: The employee will have a NULL salary.
   b. Accept the employee number as user input with a SQL*Plus substitution variable.
   c. If the employee's salary is less than $1,000, set the commission amount for the employee to 10% of the salary.
   d. If the employee's salary is between $1,000 and $1,500, set the commission amount for the employee to 15% of the salary.
   e. If the employee's salary exceeds $1,500, set the commission amount for the employee to 20% of the salary.
   f. If the employee's salary is NULL, set the commission amount for the employee to 0.
   g. Commit.
   h. Test the PL/SQL block for each case using the following test cases, and check each updated commission.

Employee Number	**Salary**	**Resulting Commission**
7369	800	80
7934	1300	195
7499	1600	320
8000	NULL	NULL

Practice 19 (continued)

```
EMPNO ENAME    SAL       COMM
----- ------ ----- ---------
 8000 DOE                   0
 7499 ALLEN   1600        320
 7934 MILLER  1300        195
 7369 SMITH    800         80
```

If you have time, complete the following exercises.

3. Modify *p16q4.sql* to insert the text "Number is odd" or "Number is even," depending on whether the value is odd or even, into the MESSAGES table. Query the MESSAGES table to determine if your PL/SQL block worked.

```
RESULTS
---------------
Number is even
```

4. Add a new column to the EMP table for storing asterisk (*).
5. Create a PL/SQL block that rewards an employee by appending an asterisk in the STARS column for every $100 of the employee's salary. Round the employee's salary to the nearest whole number. Save your PL/SQL block to a file called *p19q5.sql.*
   a. Accept the employee ID as user input with a SQL*Plus substitution variable.
   b. Initialize a variable to contain a string of asterisks.
   c. Append an asterisk to the string for every $100 of the salary amount. For example, if the employee has a salary amount of $800, the string of asterisks should contain eight asterisks.
   d. Update the STARS column for the employee with the string of asterisks.
   e. Commit.
   f. Test the block for employees who have no salary and for an employee who has a salary.

Practice 19 (continued)

```
Please enter the employee number: 7934
PL/SQL procedure successfully completed.
Please enter the employee number: 8000
PL/SQL procedure successfully completed.

EMPNO    SAL STARS
----- ------ ----------------
 8000
 7934   1300 *************
```

20

Working with Composite Datatypes

Objectives

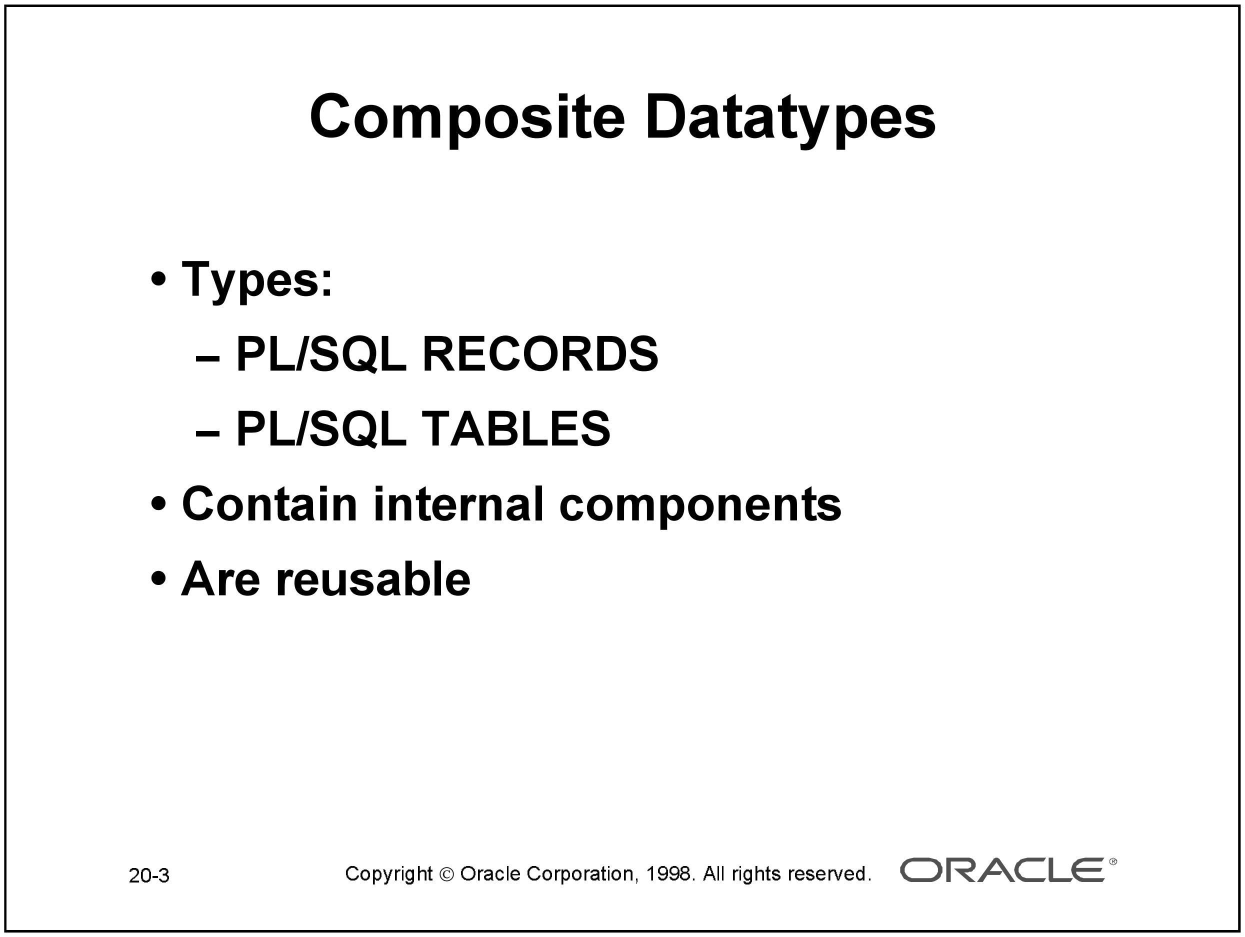
After completing this lesson, you should be able to do the following:

- Create user-defined PL/SQL records
- Create a record with the %ROWTYPE attribute
- Create a PL/SQL table
- Create a PL/SQL table of records
- Describe the difference between records, tables, and tables of records

20-2

Lesson Aim

In this lesson, you will learn more about composite datatypes and their uses.



Composite Datatypes

- Types:
  - PL/SQL RECORDS
  - PL/SQL TABLES
- Contain internal components
- Are reusable

20-3

RECORDS and TABLES

Composite datatypes (also known as *collections*) are RECORD, TABLE, Nested TABLE, and VARRAY. You use the RECORD datatype to treat related but dissimilar data as a logical unit. You use the TABLE datatype to reference and manipulate collections of data as a whole object. The Nested TABLE and VARRAY datatypes are not covered in this course.

A record is a group of related data items stored in fields, each with its own name and datatype. A Table contains a column and a primary key to give you array-like access to rows. Once defined, tables and records can be reused.

For more information, see
PL/SQL User's Guide and Reference, Release 8, "Collections and Records."

PL/SQL Records

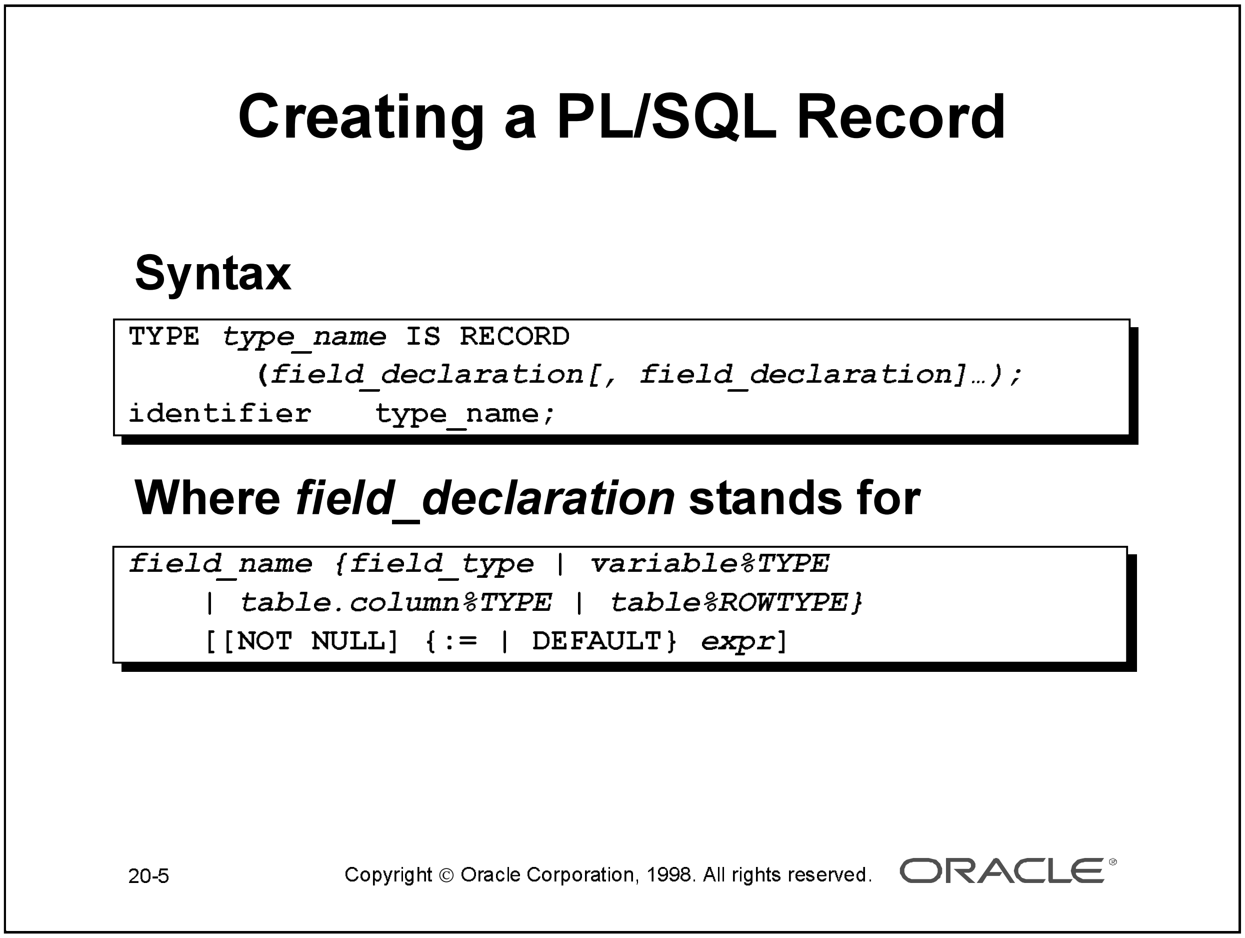
- Must contain one or more components of any scalar, RECORD, or PL/SQL TABLE datatype-called fields
- Are similar in structure to records in a 3GL
- Are not the same as rows in a database table
- Treat a collection of fields as a logical unit
- Are convenient for fetching a row of data from a table for processing

20-4 ORACLE®

PL/SQL Records

A *record* is a group of related data items stored in *fields,* each with its own name and datatype. For example, suppose you have different kinds of data about an employee such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such fields as the name, salary, and hire date of an employee lets you treat the data as a logical unit. When you declare a record type for these fields, they can be manipulated as a unit.

- Each record defined can have as many fields as necessary.
- Records can be assigned initial values and can be defined as NOT NULL.
- Fields without initial values are initialized to NULL.
- The DEFAULT keyword can also be used when defining fields.
- You can define RECORD types and declare user-defined records in the declarative part of any block, subprogram, or package.
- You can declare and reference nested records. A record can be the component of another record.



Creating a PL/SQL Record

Syntax

```
TYPE type_name IS RECORD
       (field_declaration[, field_declaration]…);
identifier    type_name;
```

Where *field_declaration* stands for

```
field_name {field_type | variable%TYPE
    | table.column%TYPE | table%ROWTYPE}
    [[NOT NULL] {:= | DEFAULT} expr]
```

20-5 ORACLE®

Defining and Declaring a PL/SQL Record

To create a record, you define a RECORD type and then declare records of that type.

In the syntax,

type_name	is the name of the RECORD type. This identifier is used to declare records.
field_name	is the name of a field within the record.
field_type	is the datatype of the field. It represents any PL/SQL datatype except REF CURSOR. You can use the %TYPE and %ROWTYPE attributes.
expr	is the *field_type* or and initial value.

The NOT NULL constraint prevents the assigning of nulls to those fields. Be sure to initialize NOT NULL fields.

Creating a PL/SQL Record

Declare variables to store the name, job, and salary of a new employee.

Example

```
...
  TYPE emp_record_type IS RECORD
    (ename       VARCHAR2(10),
     job         VARCHAR2(9),
     sal         NUMBER(7,2));
  emp_record     emp_record_type;
...
```

Creating a PL/SQL Record

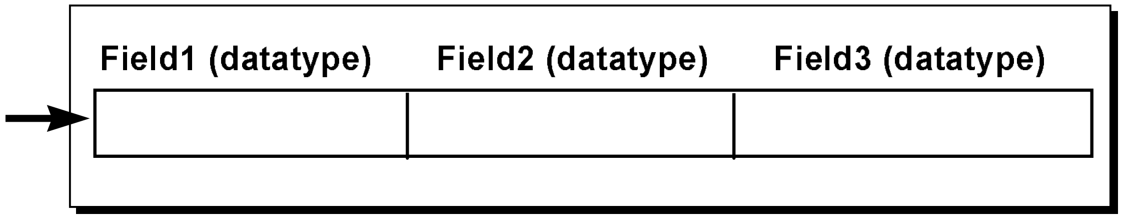
Field declarations are like variable declarations. Each field has a unique name and a specific datatype. There are no predefined datatypes for PL/SQL records, as there are for scalar variables. Therefore, you must create the datatype first and then declare an identifier using that datatype.

The following example shows that you can use the %TYPE attribute to specify a field datatype:

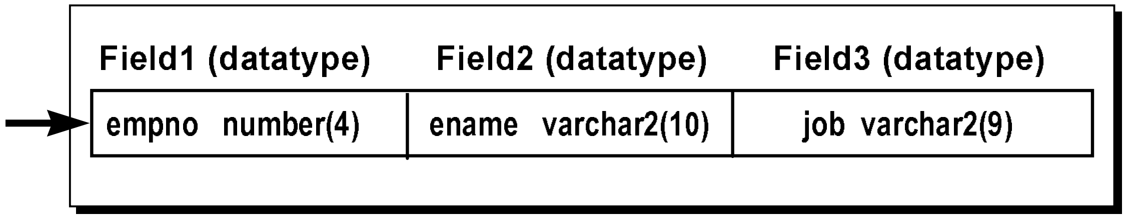
```
DECLARE
  TYPE emp_record_type IS RECORD
    (empno      NUMBER(4) NOT NULL := 100,
     ename      emp.ename%TYPE,
     job        emp.job%TYPE);
  emp_record    emp_record_type;
...
```

Note: You can add the NOT NULL constraint to any field declaration and so prevent the assigning of nulls to that field. Remember, fields declared as NOT NULL must be initialized.

PL/SQL Record Structure



Example



Referencing and Initializing Records

Fields in a record are accessed by name. To reference or initialize an individual field, you use dot notation and the following syntax:

```
record_name.field_name
```

For example, you reference field *job* in record *emp_record* as follows:

```
emp_record.job ...
```

You can then assign a value to the record field as follows:

```
emp_record.job := 'CLERK';
```

In a block or subprogram, user-defined records are instantiated when you enter the block or subprogram and cease to exist when you exit the block or subprogram.

Assigning Values to Records

You can assign a list of common values to a record by using the SELECT or FETCH statement. Make sure that the column names appear in the same order as the fields in your record. You can also assign one record to another if they have the same datatype. A user-defined record and a %ROWTYPE record *never* have the same datatype.

The %ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table.
- Fields in the record take their names and datatypes from the columns of the table or view.

20-8 ORACLE®

Declaring Records with the %ROWTYPE Attribute

To declare a record based upon a collection of columns in a database table or view, you use the %ROWTYPE attribute. The fields in the record take their names and datatypes from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

In the following example a record is declared using %ROWTYPE as a datatype specifier.

```
DECLARE
   emp_record       emp%ROWTYPE;
```

The record, *emp_record,* will consist of the following fields, which reflect all of the fields that belong to the EMP table.

```
(ename      VARCHAR2(10),
 job        VARCHAR2(9),
 sal        NUMBER,
 comm       NUMBER)
```

.

Advantages of Using %ROWTYPE

- The number and datatypes of the underlying database columns may not be known.
- The number and datatypes of the underlying database column may change at runtime.
- Useful when retrieving a row with the SELECT statement.

Declaring Records with the %ROWTYPE Attribute (continued)

```
DECLARE
   identifier      reference%ROWTYPE;
```

where:	*identifier*	is the name chosen for the record as a whole.
	reference	is the name of the table, view, cursor, or cursor variable on which the record is to be based. You must make sure that this reference is valid when you declare the record (that is, the table or view must exist).

To reference an individual field, you use dot notation and the following syntax:

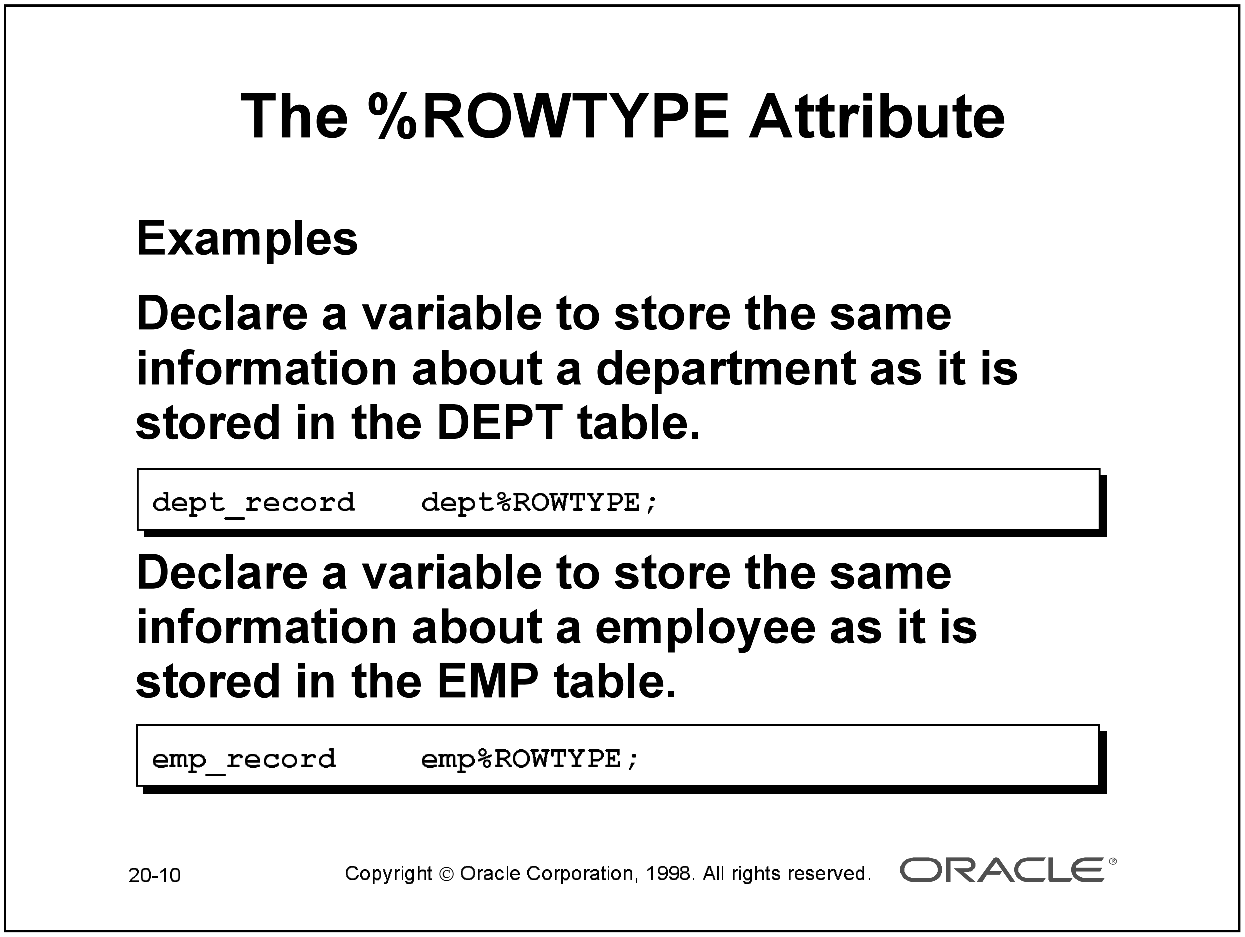
```
record_name.field_name
```

For example, you reference field *grade* in record *salary_profile* as follows:

```
emp_record.comm
```

You can then assign a value to the record field as follows:

```
emp_record.sal := 75000;
```



The %ROWTYPE Attribute

Examples

Declare a variable to store the same information about a department as it is stored in the DEPT table.

```
dept_record    dept%ROWTYPE;
```

Declare a variable to store the same information about a employee as it is stored in the EMP table.

```
emp_record     emp%ROWTYPE;
```

20-10

Examples

The first declaration in the slide creates a record with the same field names and field datatypes as a row in the DEPT table. The fields are DEPTNO, DNAME, and LOCATION.

The second declaration above creates a record with the same field names and field datatypes as a row in the EMP table. The fields are EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, and DEPTNO.

In the following example, you select column values into a record named *item_record.*

```
DECLARE
   item_record   item%ROWTYPE;
   ...
BEGIN
   SELECT * INTO item_record
   FROM   item
   WHERE  ...
```

PL/SQL Tables

- Are composed of two components:
  - Primary key of datatype BINARY_INTEGER
  - Column of scalar or record datatype
- Increase dynamically because they are unconstrained

20-11

PL/SQL Tables

Objects of type TABLE are called PL/SQL tables. They are modeled as (but not the same as) database tables. PL/SQL tables use a primary key to give you array like access to rows.

A PL/SQL table:

- Is similar to an array
- Must contain two components:
  - A primary key of datatype BINARY_INTEGER that indexes the PL/SQL TABLE.
  - A column of a scalar or record datatype, which stores the PL/SQL TABLE elements.
- Can increase dynamically because it is unconstrained

Creating a PL/SQL Table

Syntax

```
TYPE type_name IS TABLE OF
  {column_type | variable%TYPE
  | table.column%TYPE} [NOT NULL]
  [INDEX BY BINARY_INTEGER];
identifier   type_name;
```

Declare a PL/SQL variable to store a name.

Example

```
...
TYPE ename_table_type IS TABLE OF emp.ename%TYPE
  INDEX BY BINARY_INTEGER;
ename_table ename_table_type;
...
```

20-12 ORACLE®

Declaring PL/SQL Tables

There are two steps involved in creating a PL/SQL table.

1. Declare a TABLE datatype.
2. Declare a variable of that datatype.

In the syntax,

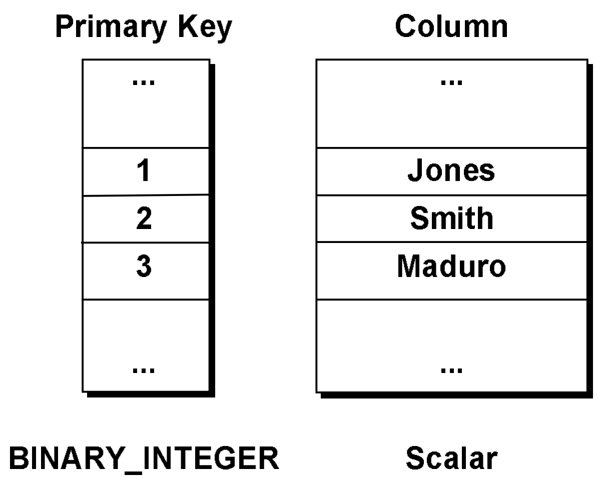
type_name	is the name of the TABLE type. It is a type specifier used in subsequent declarations of PL/SQL tables.
column_type	is any scalar (not composite) datatype such as VARCHAR2, DATE, or NUMBER. You can use the %TYPE attribute to provide the column datatype.
identifier	is the name of the identifier which represents an entire PL/SQL table.

The NOT NULL constraint prevents nulls from being assigned to the PL/ SQL TABLE of that type. Do not initialize the PL/SQL TABLE.

Declare a PL/SQL variable to store the date.

```
DECLARE
  TYPE date_table_type IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
  date_table  date_table_type;
```

PL/SQL Table Structure



PL/SQL Table Structure

Like the size of a database table, the size of a PL/SQL table is unconstrained. That is, the number of rows in a PL/SQL table can increase dynamically, so your PL/SQL table grows as new rows are added.

PL/SQL tables can have one column and a primary key, neither of which can be named. The column can belong to any scalar or record data type, but the primary key must belong to type BINARY_INTEGER. You cannot initialize a PL/SQL table in its declaration.

Creating a PL/SQL Table

```
DECLARE
  TYPE ename_table_type IS TABLE OF emp.ename%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
  ename_table     ename_table_type;
  hiredate_table  hiredate_table_type;
BEGIN
  ename_table(1) := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
      INSERT INTO ...
    ...
END;
```

20-14 ORACLE®

Creating a PL/SQL Table

There are no predefined datatypes for PL/SQL records, as there are for scalar variables. Therefore you must create the datatype first and then declare an identifier using that datatype.

Referencing a PL/SQL table

Syntax

```
pl/sql_table_name(primary_key_value)
```

where: *primary_key_value* belongs to type BINARY_INTEGER.

Reference the third row in a PL/SQL table ename_table.

```
ename_table(3) ...
```

The magnitude range of a BINARY_INTEGER is -2147483647 ... 2147483647, so the primary key value can be negative. Indexing need not start with 1.

Note: The *table*.EXISTS(i) statement returns TRUE if at least one row with index *i* is returned. Use the EXISTS statement to prevent an error which is raised in reference to a non-existing table element.

PL/SQL Table of Records

- **Define a TABLE variable with the %ROWTYPE attribute.**
- **Declare a PL/SQL variable to hold department information.**

Example

```
DECLARE
  TYPE dept_table_type IS TABLE OF dept%ROWTYPE
    INDEX BY BINARY_INTEGER;
  dept_table dept_table_type;
  -- Each element of dept_table is a record
```

20-15 ORACLE®

PL/SQL Table of Records

Because only one table definition is needed to hold information about all of the fields of a database table, the table of records greatly increases the functionality of PL/SQL tables.

Referencing a Table of Records

In the example given in the slide, you can refer to fields in the dept_table record because each element of this table is a record.

Syntax

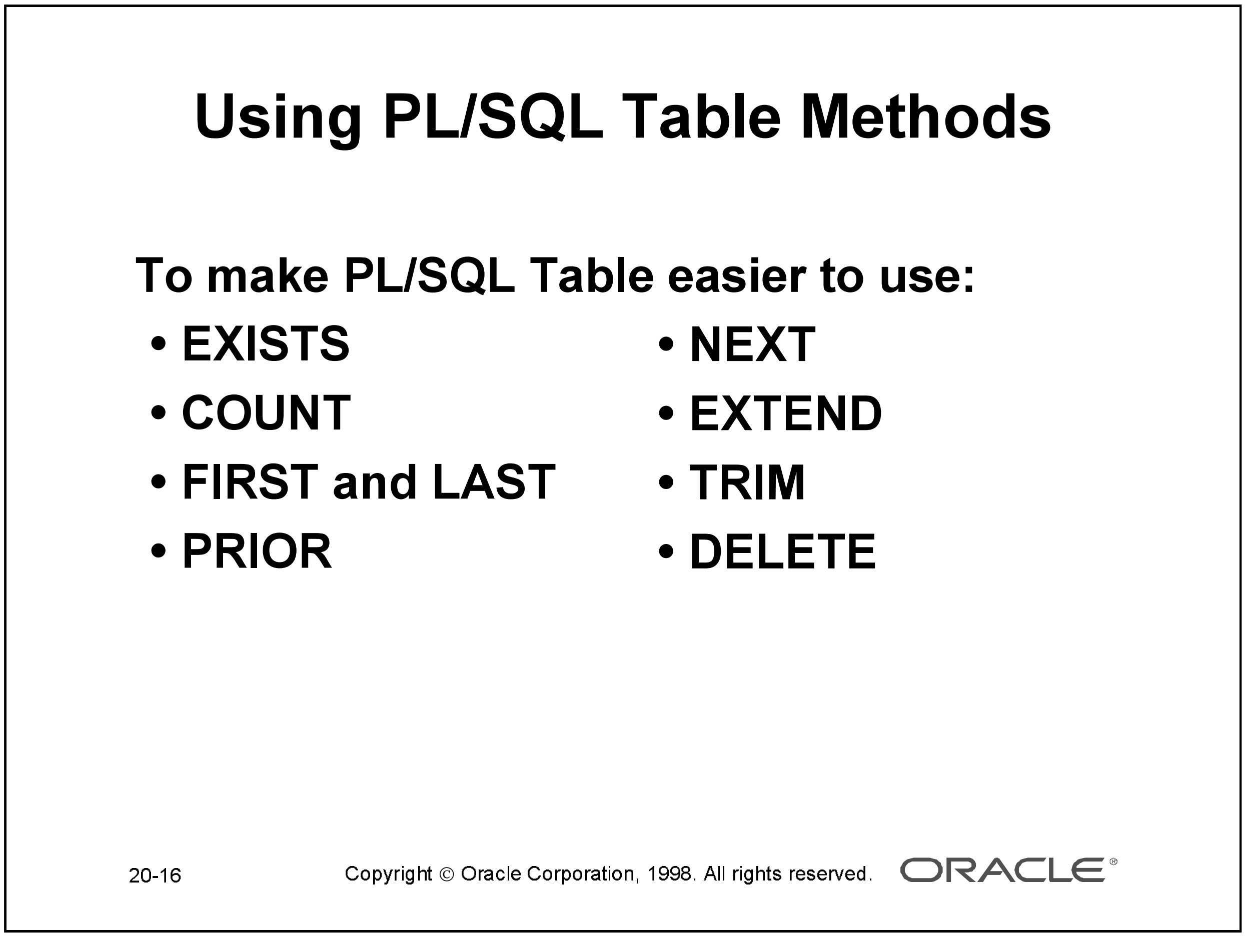
```
table(index).field
```

Example

```
dept_table(15).location := 'Atlanta';
```

Location represents a field in the DEPT table.

Note: You can use the %ROWTYPE attribute to declare a record that represents a row in a database table. The difference between the %ROWTYPE attribute and the composite datatype RECORD is that RECORD allows you to specify the datatypes of fields in the record or to declare fields of your own.



Using PL/SQL Table Methods

To make PL/SQL Table easier to use:

- EXISTS
- COUNT
- FIRST and LAST
- PRIOR
- NEXT
- EXTEND
- TRIM
- DELETE

A PL/SQL Table method is a built-in procedure or function that operates on tables and is called using dot notation.

Syntax:

```
table_name.method_name[ (parameters) ]
```

Method	**Description**
EXISTS(n)	Returns TRUE if the *n*th element in a PL/SQL Table exists.
COUNT	Returns the number of elements that a PL/SQL Table currently contains.
FIRST LAST	Return the first and last (smallest and largest) index numbers in a PL/SQL Table. Returns NULL if the PL/SQL Table is empty.
PRIOR(n)	Returns the index number that precedes index *n* in a PL/SQL Table.
NEXT(n)	Returns the index number that succeeds index *n* in a PL/SQL Table.
EXTEND(n, i)	To increase the size of a PL/SQL Table. EXTEND appends one null element to a PL/SQL Table. EXTEND(n) appends *n* null elements to a PL/SQL Table. EXTEND(n, i) appends *n* copies of the *i*th element to a PL/SQL Table.
TRIM	TRIM removes one element from the end of a PL/SQL Table. TRIM(n) removes *n* elements from the end of a PL/SQL Table.
DELETE	DELETE removes all elements from a PL/SQL Table. DELETE(n) removes the *n*th element from a PL/SQL Table. DELETE(m, n) removes all elements in the range *m* ... *n* from a PL/SQL Table.

Summary

- Define and reference PL/SQL variables of composite datatypes:
  - PL/SQL Records
  - PL/SQL Tables
  - PL/SQL Table of Records
- Define a PL/SQL Record using the %ROWTYPE attribute.

20-17

Practice Overview

- Declaring PL/SQL tables
- Processing data using PL/SQL tables

20-18

Practice Overview

In this practice, you define, create, and use PL/SQL tables.

Practice 20

1. Run the script LABS*lab20_1.sql* to create a new table for storing employees and their salaries.

```
SQL> CREATE TABLE     top_dogs
  2  (name            VARCHAR2(25),
  3   salary          NUMBER(11,2));
```

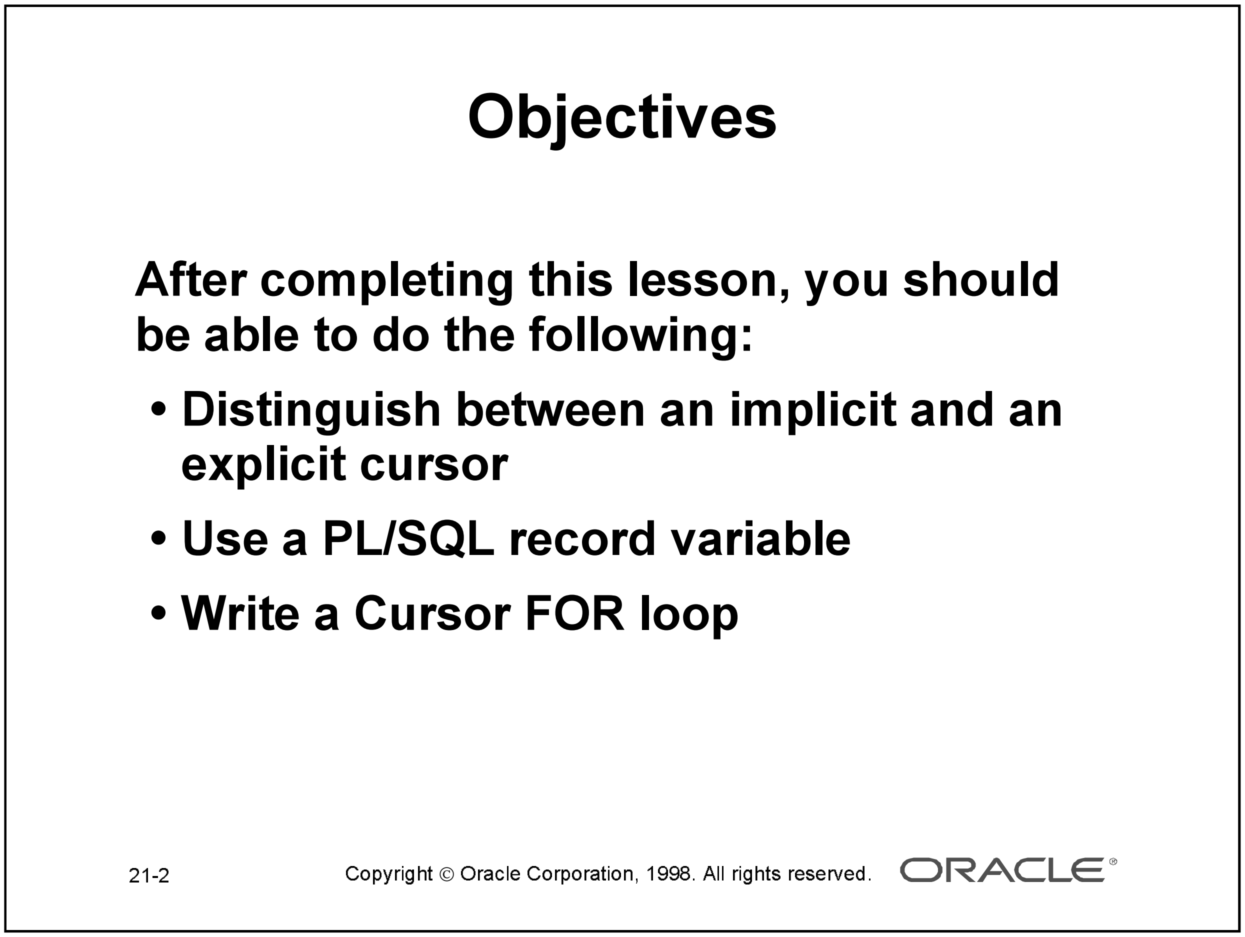
2. Write a PL/SQL block to retrieve the name and salary of a given employee from the EMP table based on the employee's number, incorporate PL/SQL tables.
   a. Declare two PL/SQL tables, ENAME_TABLE and SAL_TABLE, to temporarily store the names and salaries.
   b. As each name and salary is retrieved within the loop, store them in the PL/SQL tables.
   c. Outside the loop, transfer the names and salaries from the PL/SQL tables into the TOP_DOGS table.
   d. Empty the TOP_DOGS table and test the practice.

```
Please enter the employee number: 7934
PL/SQL procedure successfully completed.
NAME               SALARY
--------------- ---------
MILLER               1300


Please enter the employee number: 7876
PL/SQL procedure successfully completed.
NAME               SALARY
--------------- ---------
ADAMS               1100
```

21

Writing Explicit Cursors



Objectives

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

- **Distinguish between an implicit and an explicit cursor**
- **Use a PL/SQL record variable**
- **Write a Cursor FOR loop**

21-2

Lesson Aim

In this lesson you will learn the differences between implicit and explicit cursors. You will also learn when and why to use an explicit cursor.

You may need to use a multiple row SELECT statement in PL/SQL to process many rows. To accomplish this, you declare and control explicit cursors, which are used in loops, including the cursor FOR loop.

About Cursors

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

- **Implicit cursors: Declared for all DML and PL/SQL SELECT statements.**
- **Explicit cursors: Declared and named by the programmer.**

Implicit and Explicit Cursors

The Oracle Server uses work areas called *private SQL areas* to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information. The cursor directs all phases of processing.

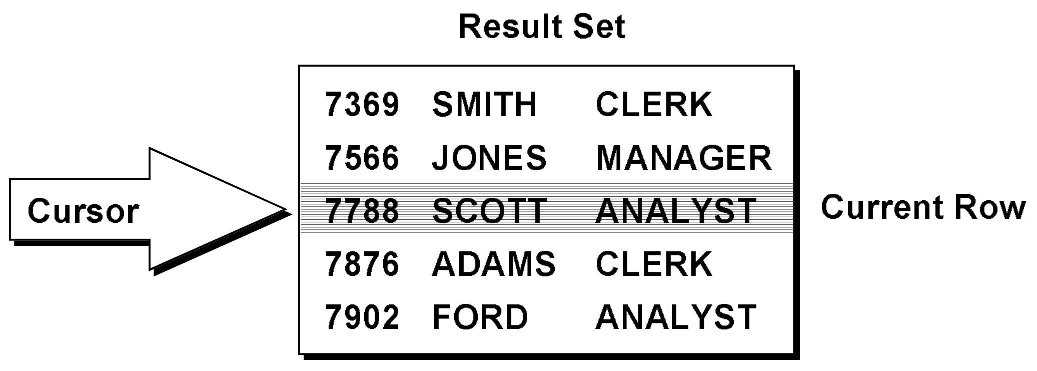
Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row.
Explicit	For queries that return more than one row. Explicit cursors are declared and named by the programmer and manipulated through specific statements in the block's executable actions.

Implicit Cursors

The Oracle Server implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor. PL/SQL lets you refer to the most recent implicit cursor as the *SQL* cursor.

You cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor, but you can use cursor attributes to get information about the most recently executed SQL statement.

Explicit Cursor Functions



7369	SMITH	CLERK
7566	JONES	MANAGER
7788	SCOTT	ANALYST
7876	ADAMS	CLERK
7902	FORD	ANALYST

21-4

Explicit Cursors

Use explicit cursors to individually process each row returned by a multi-row SELECT statement.

The set of rows returned by a multi-row query is called the *result set.* Its size is the number of rows that meet your search criteria. The diagram in the slide shows how an explicit cursor "points" to the *current row* in the result set. This allows your program to process the rows one at a time.

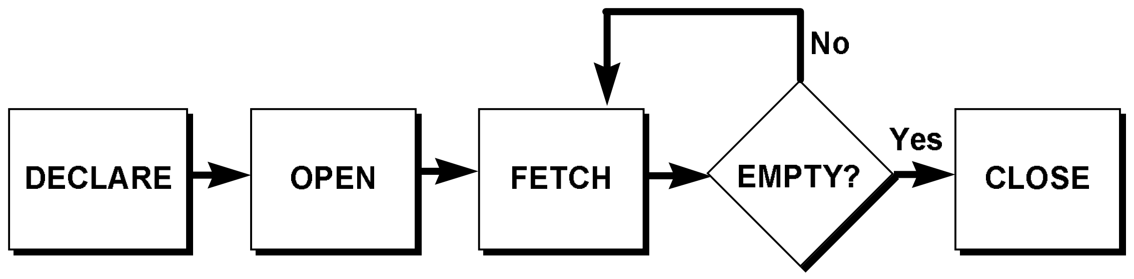
A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in a result set.

Explicit Cursor Functions

- Can process beyond the first row returned by the query, row by row
- Keep track of which row is currently being processed
- Allow the programmer to manually control them in the PL/SQL block

Note: The fetch for an implicit cursor is an array fetch, and the existence of a second row still raises the TOO_MANY_ROWS exception. Furthermore, you can use explicit cursors to perform multiple fetches and to re-execute parsed queries in the work area.

Controlling Explicit Cursors



- DECLARE
  - Create a named SQL area
- OPEN
  - Identify the active set
- FETCH
  - Load the current row into variables
- EMPTY?
  - Test for existing rows
  - Return to FETCH if rows found
- CLOSE
  - Release the active set

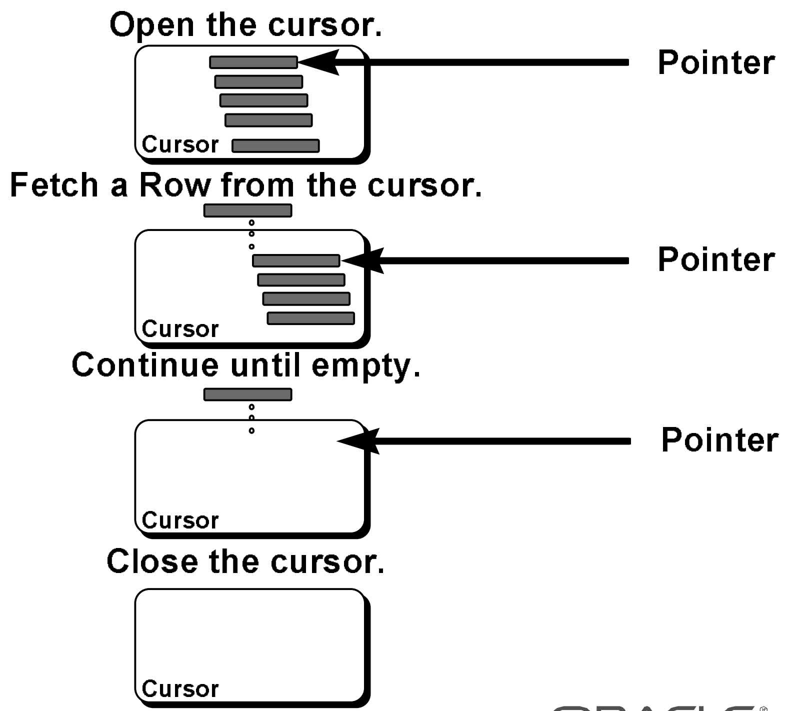
Explicit Cursors (continued)

Now that you have a conceptual understanding of cursors, review the steps to use them. The syntax for each step can be found on the following pages.

Controlling Explicit Cursors Using Four Commands

1. Declare the cursor by naming it and defining the structure of the query to be performed within it.
2. Open the cursor. The OPEN statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor. The FETCH statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore each fetch accesses a different row returned by the query. In the flow diagram shown in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise it closes the cursor.
4. Close the cursor. The CLOSE statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Controlling Explicit Cursors



Explicit Cursors (continued)

You use the OPEN, FETCH, and CLOSE statements to control a cursor. The OPEN statement executes the query associated with the cursor, identifies the result set, and positions the cursor (pointer) before the first row. The FETCH statement retrieves the current row and advances the cursor to the next row. When the last row has been processed, the CLOSE statement disables the cursor.

Declaring the Cursor

Syntax

```
CURSOR cursor_name IS
     select_statement;
```

- **Do not include the INTO clause in the cursor declaration.**
- **If processing rows in a specific sequence is required use the ORDER BY clause in the query.**

21-7

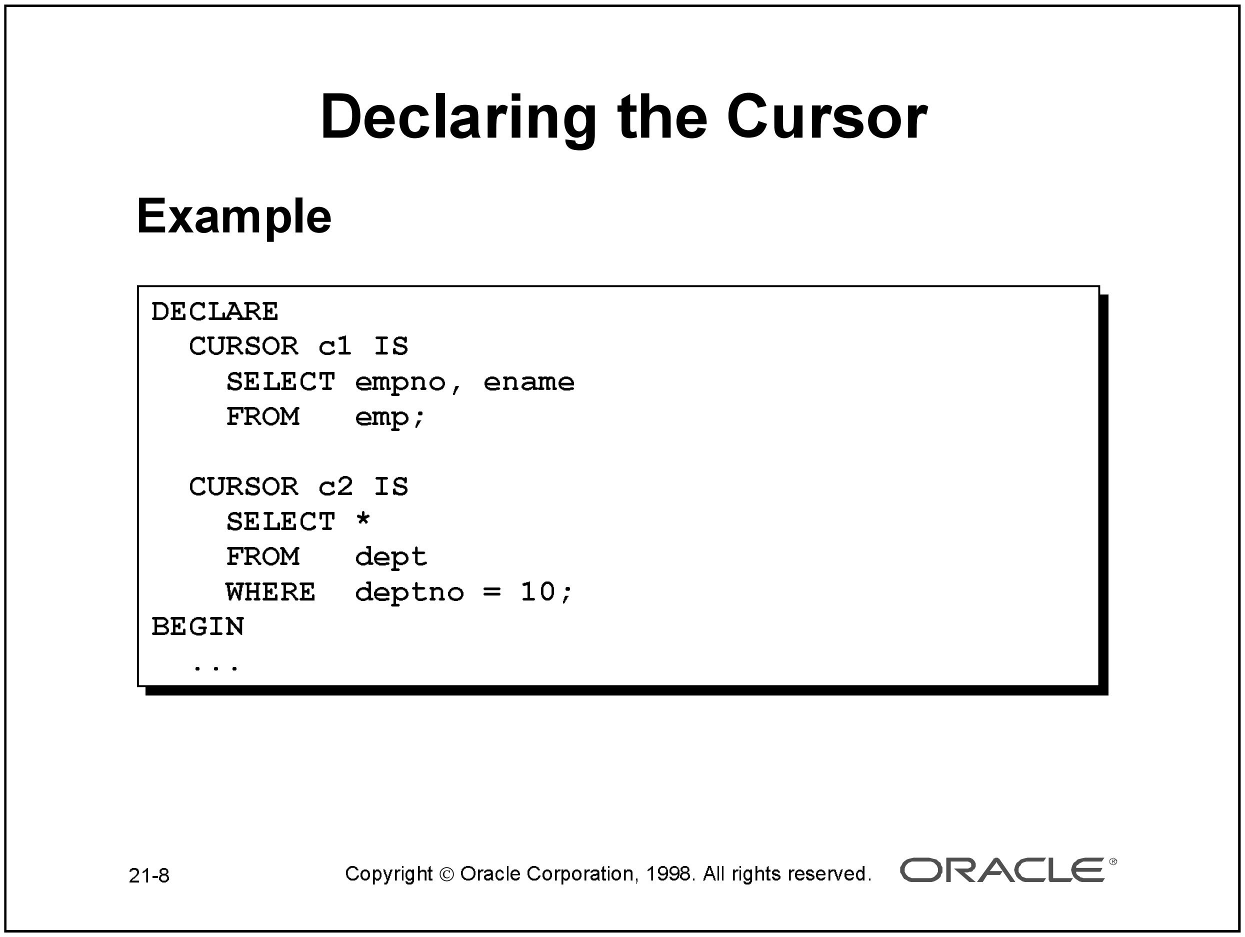
Explicit Cursor Declaration

Use the CURSOR statement to declare an explicit cursor. You can reference variables within the query, but you must declare them before the CURSOR statement.

In the syntax,

cursor_name	is a PL/SQL identifier.
select_statement	is a SELECT statement without an INTO clause.

Note: Do not include the INTO clause in the cursor declaration because it appears later in the FETCH statement.



Declaring the Cursor

Example

```
DECLARE
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;

  CURSOR c2 IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
  ...
```

21-8

Explicit Cursor Declaration (continued)

Retrieve the employees one by one.

```
DECLARE
  v_empno            emp.empno%TYPE;
  v_ename            emp.ename%TYPE;
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;
BEGIN
...
```

Note: You can reference variables in the query, but you must declare them before the CURSOR statement.

Opening the Cursor

Syntax

```
OPEN  cursor_name;
```

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

OPEN Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax,

cursor_name	is the name of the previously declared cursor.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information.
2. Parses the SELECT statement.
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses.
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.
5. Positions the pointer just before the first row in the active set.

Note: If the query returns no rows when the cursor is opened, PL/SQL does not raise an exception. However, you can test the cursor's status after a fetch.

For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows.

Fetching Data from the Cursor

Syntax

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

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

FETCH Statement

The FETCH statement retrieves the rows in the result set one at a time. After each fetch, the cursor advances to the next row in the result set.

In the syntax,

cursor_name	is the name of the previously declared cursor.
variable	is an output variable to store the results.
record_name	is the name of the record in which the retrieved data is stored. The record variable can be declared using the %ROWTYPE attribute.

Guidelines

- Include the same number of variables in the INTO clause of the FETCH statement as output columns in the SELECT statement, and be sure that the datatypes are compatible.
- Match each variable to correspond to the columns positionally.
- Alternatively, define a record for the cursor and reference the record in the FETCH INTO clause.
- Test to see if the cursor contains rows. If a fetch acquires no values, there are no rows left to process in the active set and no error is recorded.

Note: The FETCH statement performs the following operations:

1. Advances the pointer to the next row in the active set.
2. Reads the data for the current row into the output PL/SQL variables.
3. Exits the cursor FOR loop if the pointer is positioned at the end of the active set.

Fetching Data from the Cursor

Examples

```
FETCH c1 INTO v_empno, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```

21-11 ORACLE®

FETCH Statement (continued)

You use the FETCH statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the INTO list. Also, their datatypes must be compatible.

Retrieve the first ten employees one by one.

```
DECLARE
  v_empno             emp.empno%TYPE;
  v_ename             emp.ename%TYPE;
  i                   NUMBER := 1;
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;
BEGIN
  OPEN c1;
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empno, v_ename;
    ...
  END LOOP;
END ;
```

Closing the Cursor

Syntax

```
CLOSE    cursor_name;
```

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

21-12 ORACLE®

CLOSE Statement

The CLOSE statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore you can establish an active set several times.

In the syntax,

cursor_name	is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor once it has been closed, or the INVALID_CURSOR exception will be raised.

Note: The CLOSE statement releases the context area.
Although it is possible to terminate the PL/SQL block without closing cursors, you should get into the habit of closing any cursor that you declare explicitly in order to free up resources.
There is a maximum limit to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter field. OPEN_CURSORS = 50 by default.

```
...
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empno, v_ename;
    ...
  END LOOP;
  CLOSE c1;
END;
```

Explicit Cursor Attributes

Obtain status information about a cursor.

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

21-13

Explicit Cursor Attributes

As with implicit cursors, there are four attributes for obtaining status information about a cursor. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a data manipulation statement.

Note: You cannot reference cursor attributes directly in a SQL statement.

Controlling Multiple Fetches

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

Controlling Multiple Fetches from Explicit Cursors

To process several rows from an explicit cursor, you typically define a loop to perform a fetch on each iteration. Eventually all rows in the active set are processed, and an unsuccessful fetch sets the %NOTFOUND attribute to TRUE. Use the explicit cursor attributes to test the success of each fetch before any further references are made to the cursor. If you omit an exit criterion, an infinite loop results.

For more information, see
PL/SQL User's Guide and Reference, Release 8, "Interaction With Oracle."

The %ISOPEN Attribute

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

Example

```
IF NOT c1%ISOPEN THEN
   OPEN c1;
END IF;
LOOP
  FETCH c1...
```

21-15 ORACLE®

Explicit Cursor Attributes

- You can fetch rows only when the cursor is open. Use the %ISOPEN cursor attribute to determine whether the cursor is open, if necessary.
- Fetch rows in a loop. Use cursor attributes to determine when to exit the loop.
- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows, fetch the rows in a numeric FOR loop, or fetch the rows in a simple loop and determine when to exit the loop.

Note: %ISOPEN returns the status of the cursor; TRUE if open and FALSE if not. It is not usually necessary to inspect %ISOPEN.

The %NOTFOUND and %ROWCOUNT Attributes

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

21-16

Example

Retrieve the first ten employees one by one.

```
DECLARE
  v_empno            emp.empno%TYPE;
  v_ename            emp.ename%TYPE;
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO v_empno, v_ename;
    EXIT WHEN c1%ROWCOUNT > 10 OR c1%NOTFOUND;
    ...
  END LOOP;
  CLOSE c1;
END ;
```

Note: Before the first fetch, %NOTFOUND evaluates to NULL. So if FETCH never executes successfully, the loop is never exited. That is because the EXIT WHEN statement executes only if its WHEN condition is true. To be safe, you might want to use the following EXIT statement:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

If using %ROWCOUNT, add a test for no rows in the cursor by using the %NOTFOUND attribute, because the row count is not incremented if the fetch does not retrieve any rows.

Cursors and Records

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

Example

```
...
  CURSOR c1 IS
    SELECT  empno, ename
    FROM    emp;
  emp_record   c1%ROWTYPE;
BEGIN
  OPEN c1;
.  .  .
  FETCH c1 INTO emp_record;
```

21-17 ORACLE®

Cursors and Records

You have already seen that you can define records to use the structure of columns in a table. You can also define a record based on the selected list of columns in an explicit cursor. This is convenient for processing the rows of the active set, because you can simply fetch into the record. Therefore the values of the row are loaded directly into the corresponding fields of the record.

Cursor FOR Loops

Syntax

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

- **Shortcut to process explicit cursors.**
- **Implicit open, fetch, and close occur.**
- **Do not declare the record; it is implicitly declared.**

Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. It is a shortcut because the cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched.

In the syntax,

record_name	is the name of the implicitly declared record.
cursor_name	is a PL/SQL identifier for the previously declared cursor.

Guidelines

- Do not declare the record that controls the loop. Its scope is only in the loop.
- Test the cursor attributes during the loop, if required.
- Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement. More information on cursor parameters is covered in a subsequent lesson.
- Do not use a cursor FOR loop when the cursor operations must be handled manually.

Note: You can define a query at the start of the loop itself. The query expression is called a SELECT substatement, and the cursor is internal to the FOR loop. Because the cursor is not declared with a name, you cannot test its attributes.

Cursor FOR Loops

Retrieve employees one by one until there are no more left.

Example

```
DECLARE
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp;
BEGIN
  FOR emp_record IN c1 LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.empno = 7839 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Cursor FOR Loops Using Subqueries

No Need to declare the cursor.

Example

```
BEGIN
  FOR emp_record IN ( SELECT empno, ename
                      FROM   emp) LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.empno = 7839 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Cursor FOR Loops Using Subqueries

You do not need to declare a cursor because PL/SQL lets you substitute a subquery.

Summary

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

Practice Overview

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

Practice Overview

This practice applies your knowledge of cursors to process a number of rows from a table and populate another table with the results using a cursor FOR loop.

Practice 21

1. Create a PL/SQL block that determines the top employees with respect to salaries.
   a. Accept a number *n* as user input with a SQL*Plus substitution parameter.
   b. In a loop, get the last names and salaries of the top *n* people with respect to salary in the EMP table.
   c. Store the names and salaries in the TOP_DOGS table.
   d. Assume that no two employees have the same salary.
   e. Test a variety of special cases, such a *n* = 0, where *n* is greater than the number of employees in the EMP table. Empty the TOP_DOGS table after each test.

```
Please enter the number of top money makers: 5
NAME        SALARY
----------- ------
KING          5000
FORD          3000
SCOTT         3000
JONES         2975
BLAKE         2850
```

2. Consider the case where several employees have the same salary. If one person is listed, then all people who have the same salary should also be listed.
   a. For example, if the user enters a value of 2 for *n*, then King, Ford and Scott should be displayed. (These employees are tied for second highest salary.)
   b. If the user enters a value of 3, then King, Ford, Scott, and Jones should be displayed.
   c. Delete all rows from TOP_DOGS and test the practice.

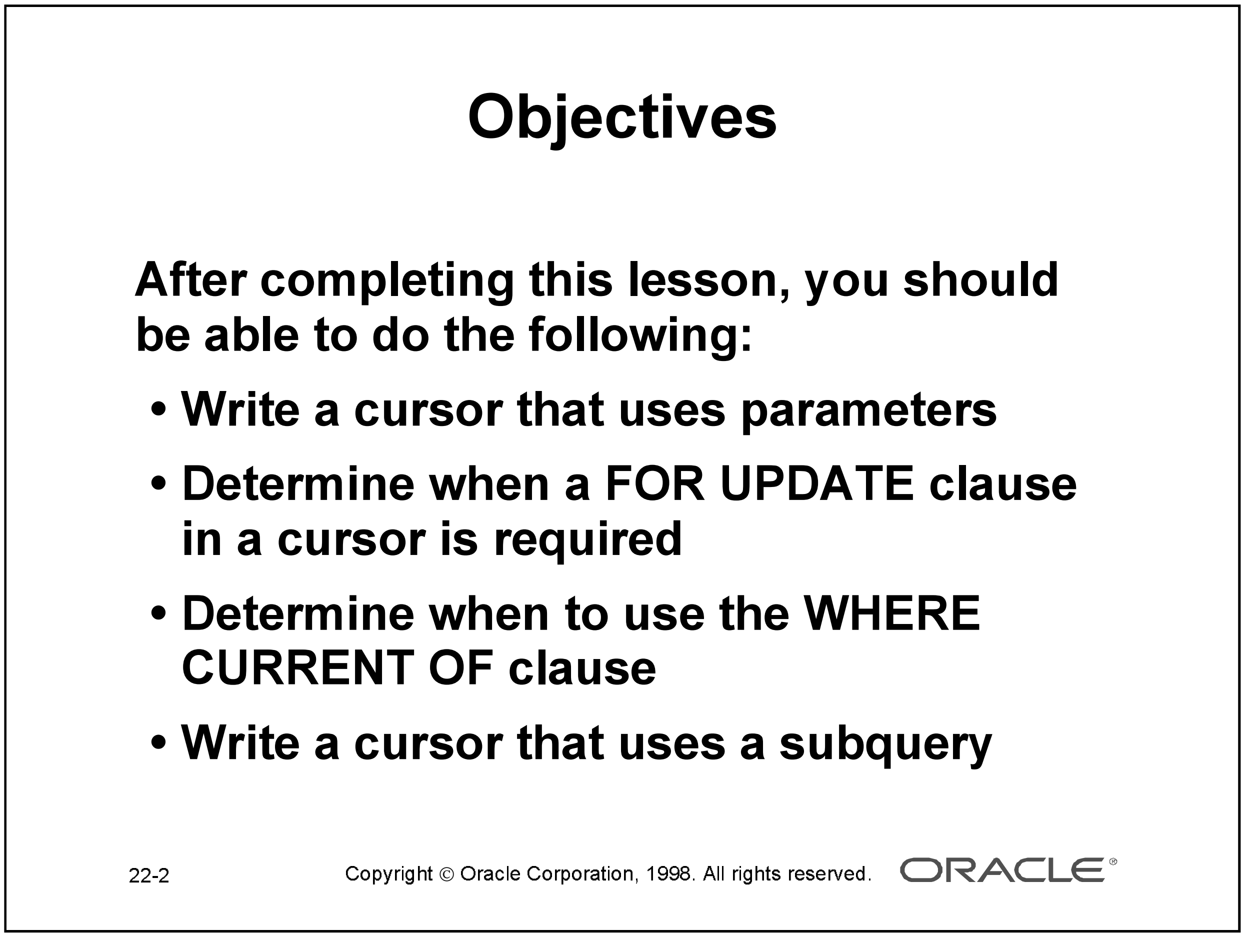
```
Please enter the number of top money makers: 2
NAME         SALARY
------------ ------
KING           5000
FORD           3000
SCOTT          3000
```

Practice 21 (continued)

```
Please enter the number of top money makers: 3
NAME         SALARY
------------ ------
KING           5000
FORD           3000
SCOTT          3000
JONES          2975
```

22

Advanced Explicit Cursor Concepts



Objectives

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

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

Lesson Aim

In this lesson you will learn more about writing explicit cursors, specifically about writing cursors that use parameters, and the PL/SQL table of records.

Cursors with Parameters

Syntax

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

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

22-3 ORACLE®

Cursors with Parameters

Parameters allow values to be passed to a cursor when it is opened and to be used in the query when it executes. This means that you can open and close an explicit cursor several times in a block, returning a different active set on each occasion.

Each formal parameter in the cursor declaration must have a corresponding actual parameter in the OPEN statement. Parameter datatypes are the same as those for scalar variables, but you do not give them sizes. The parameter names are for references in the cursor's query expression.

In the syntax,

cursor_name	is a PL/SQL identifier for the previously declared cursor.
parameter_name	is the name of a parameter. Parameter stands for the following syntax.

```
cursor_parameter_name  [IN]  datatype  [{:= | DEFAULT} expr]
```

datatype	is a scalar datatype of the parameter.
select_statement	is a SELECT statement without the INTO clause.

When the cursor is opened, you pass values to each of the parameters positionally. You can pass values from PL/SQL or host variables as well as from literals.

Note: The parameter notation does not offer greater functionality; it simply allows you to specify input values easily and clearly. This is particularly useful when the same cursor is referenced repeatedly.

Cursors with Parameters

Pass the department number and job title to the WHERE clause.

Example

```
DECLARE
  CURSOR c1
  (v_deptno NUMBER, v_job VARCHAR2) IS
    SELECT   empno, ename
    FROM     emp
    WHERE    deptno = v_deptno
     AND     job = v_job;
BEGIN
  OPEN c1(10, 'CLERK');
...
```

22-4 ORACLE®

Parameter datatypes are the same as those for scalar variables, but you do not give them sizes. The parameter names are for references in the cursor's query.

In the following example, two variables and a cursor are declared. The cursor is defined with two parameters.

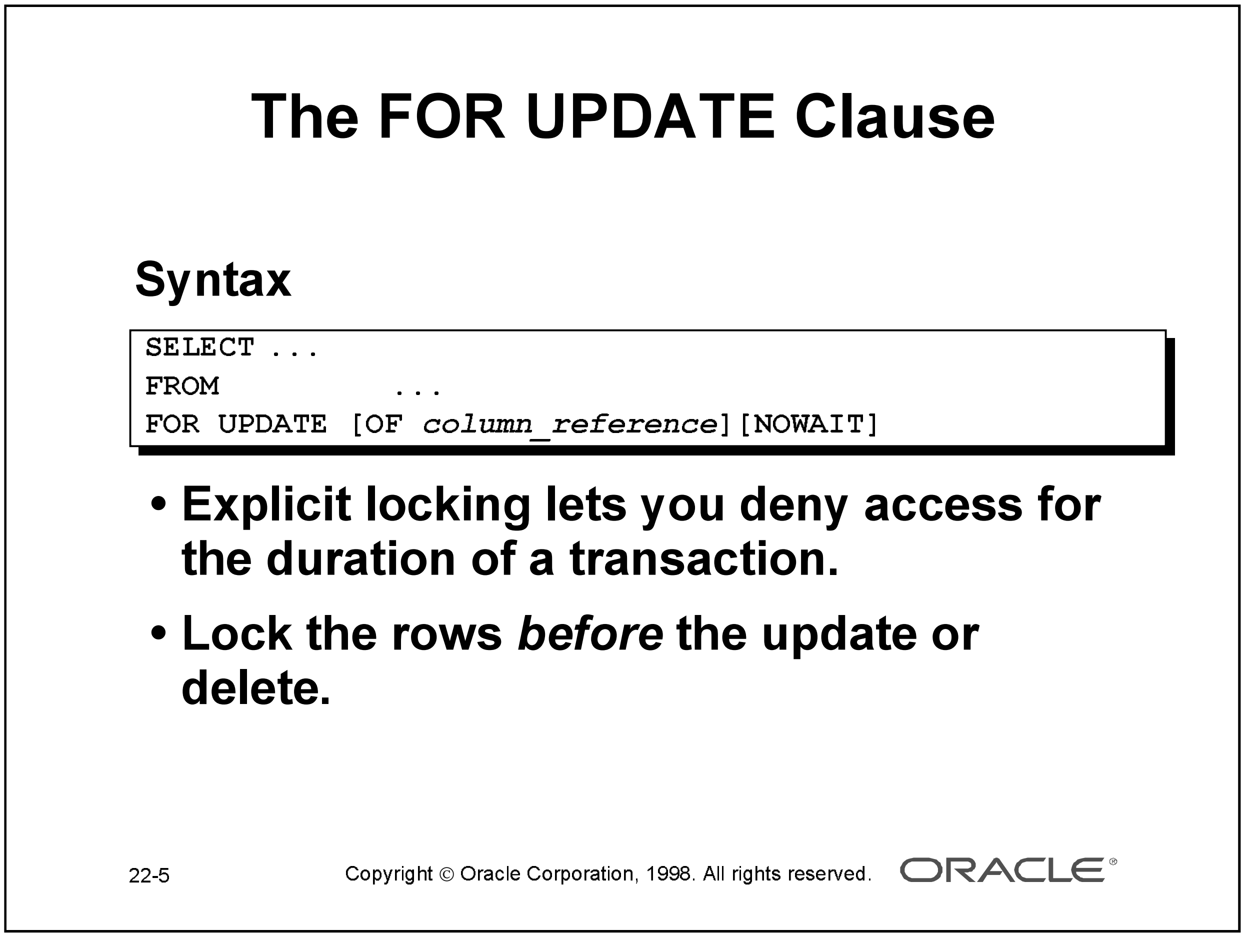
```
DECLARE
   job_emp   emp.job%TYPE := 'CLERK';
   v_ename   emp.ename%TYPE;
   CURSOR c1 (v_deptno NUMBER, v_job VARCHAR2) is
      SELECT ...
```

Either of the following statements opens the cursor:

```
OPEN c1(10, job_emp);
OPEN c1(20, 'ANALYST');
```

You can pass parameters to the cursor used in a cursor FOR loop:

```
DECLARE
   CURSOR c1 (v_deptno NUMBER, v_job VARCHAR2) is
      SELECT ...
BEGIN
   FOR emp_record IN c1(10, 'ANALYST') LOOP ...
```



The FOR UPDATE Clause

Syntax

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

- **Explicit locking lets you deny access for the duration of a transaction.**
- **Lock the rows *before* the update or delete.**

The FOR UPDATE Clause

You may want to lock rows before you update or delete rows. Add the FOR UPDATE clause in the cursor query to lock the affected rows when the cursor is opened. Because the Oracle Server releases locks at the end of the transaction, you should not commit across fetches from an explicit cursor if FOR UPDATE is used.

In the syntax,

column_reference	is a column in the table against which the query is performed. A list of columns may also be used.
NOWAIT	returns an Oracle error if the rows are locked by another session.

The FOR UPDATE clause is the last clause in a select statement, even after the order by, if one exists.

When querying multiple tables, you can use the FOR UPDATE clause to confine row locking to particular tables. Rows in a table are locked only if the FOR UPDATE clause refers to a column in that table.

Exclusive row locks are taken on the rows in the result set before the OPEN returns when the FOR UPDATE clause is used.

The FOR UPDATE Clause

Retrieve the orders for amounts over $1000 that were processed today.

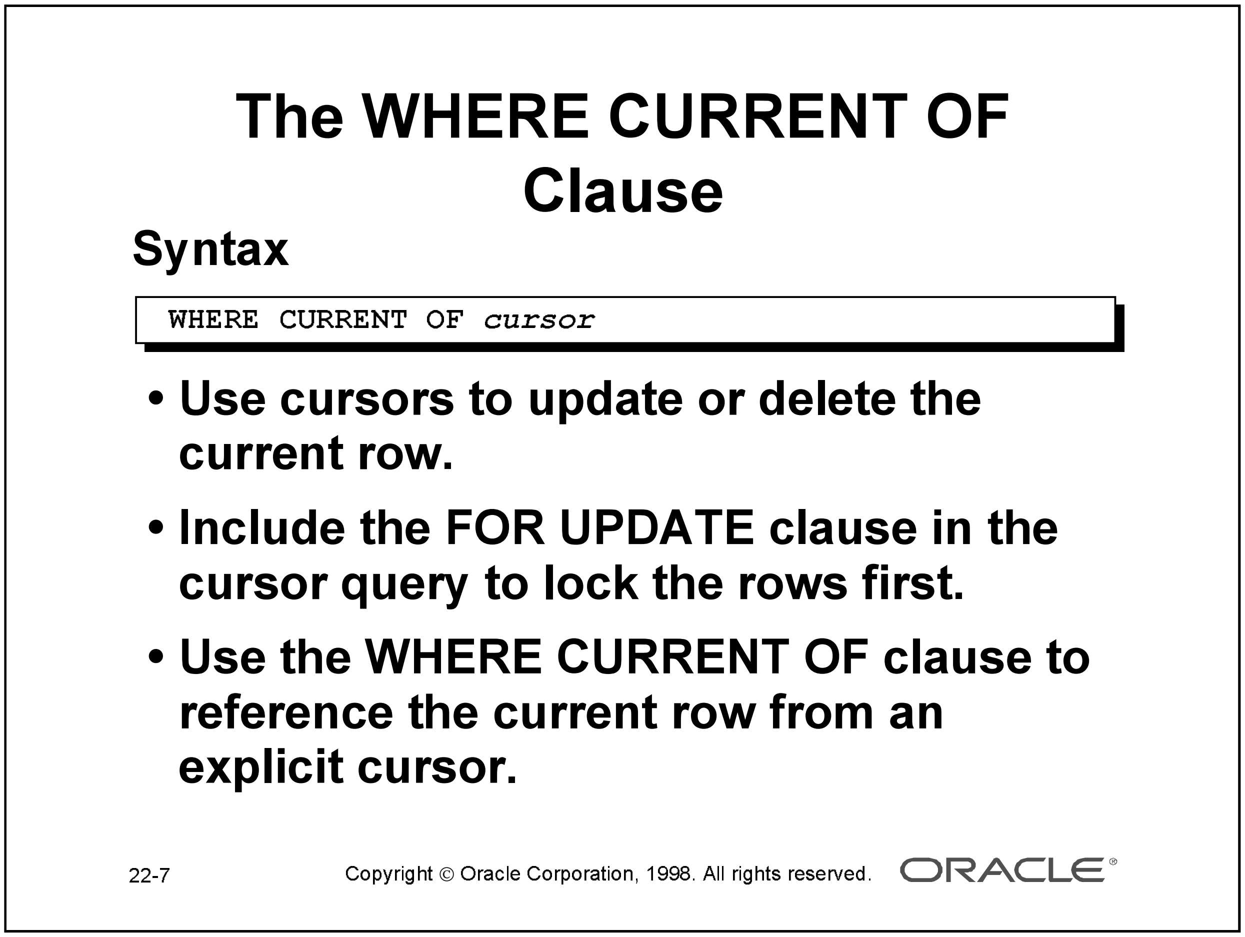
Example

```
DECLARE
  CURSOR c1 IS
    SELECT empno, ename
    FROM   emp
    FOR UPDATE NOWAIT;
```

The FOR UPDATE Clause

Note: If the Oracle Server cannot acquire the locks on the rows it needs in a SELECT FOR UPDATE, it waits indefinitely. You can use the NOWAIT clause in the SELECT FOR UPDATE statement and test for the error code that returns due to failure to acquire the locks in a loop. Therefore you can retry opening the cursor *n* times before terminating the PL/SQL block. If you have a large table, you can achieve better performance by using the LOCK TABLE statement to lock all rows in the table. However, when using LOCK TABLE, you cannot use the WHERE CURRENT OF clause and must use the notation WHERE *column* = *identifier*.

It is not mandatory that the FOR UPDATE OF clause refers to a column however it is recommended to do so for better readability and maintenance.



The WHERE CURRENT OF Clause

Syntax

```
WHERE CURRENT OF cursor
```

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

22-7 ORACLE®

The WHERE CURRENT OF Clause

When referencing the current row from an explicit cursor, use the WHERE CURRENT OF clause. This allows you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference ROWID. You must include the FOR UPDATE clause in the cursor's query so that the rows are locked on OPEN.

In the syntax,

cursor	is the name of a declared cursor. The cursor must have been declared with the FOR UPDATE clause.

The WHERE CURRENT OF Clause

Example

```
DECLARE
  CURSOR c1 IS
    SELECT ...
    FOR UPDATE NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
    ...
  END LOOP;
  COMMIT;
END;
```

22-8 ORACLE®

The WHERE CURRENT OF Clause

You can update rows based on criteria from a cursor.

Additionally, you can write your DELETE or UPDATE statement to contain the WHERE CURRENT OF *cursor_name* clause to refer to the latest row processed by the FETCH statement. When you use this clause, the cursor you reference must exist and must contain the FOR UPDATE clause in the cursor query; otherwise you will obtain an error. This clause allows you to apply updates and deletes to the currently addressed row without the need to explicitly reference the ROWID pseudo-column.

Cursors with Subqueries

Example

```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.deptno, dname,  STAFF
    FROM   dept t1, (SELECT   deptno,
                              count(*) STAFF
                     FROM     emp
                     GROUP BY deptno) t2
    WHERE  t1.deptno = t2.deptno
    AND    STAFF >= 5;
```

Subqueries

A subquery is a query (usually enclosed by parentheses) that appears within another SQL data manipulation statement. When evaluated, the subquery provides a value or set of values to the statement.

- Subqueries are often used in the WHERE clause of a select statement. They may also be used in the FROM clause, as in the example.
- A subquery or correlated subquery may be used.

Summary

- Return different results sets using cursors with parameters.
- Define cursors with subqueries and correlated subqueries.
- Manipulate explicit cursors with commands:
  - FOR UPDATE Clause
  - WHERE CURRENT OF Clause

22-10

Practice Overview

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

Practice Overview

This practice applies your knowledge of cursors with parameters to process a number of rows from multiple tables.

Practice 22

1. Write a query to retrieve all the departments and the employees in each department. Insert the results in the MESSAGES table. Use a cursor to retrieve the department number and pass the department number to a cursor to retrieve the employees in that department.

```
RESULTS
------------------------
KING - Department 10
CLARK - Department 10
MILLER - Department 10
JONES - Department 20
FORD - Department 20
SMITH - Department 20
SCOTT - Department 20
ADAMS - Department 20
BLAKE - Department 30
MARTIN - Department 30
ALLEN - Department 30
TURNER - Department 30
JAMES - Department 30
WARD - Department 30
14 rows selected.
```

2. Modify *p19q5.sql* to incorporate the FOR UPDATE and WHERE CURRENT OF functionality in cursor processing.

```
EMPNO    SAL STARS
----- ------ ---------------------
 8000
 7900    950 **********
 7844   1500 ***************
```

23

Handling Exceptions

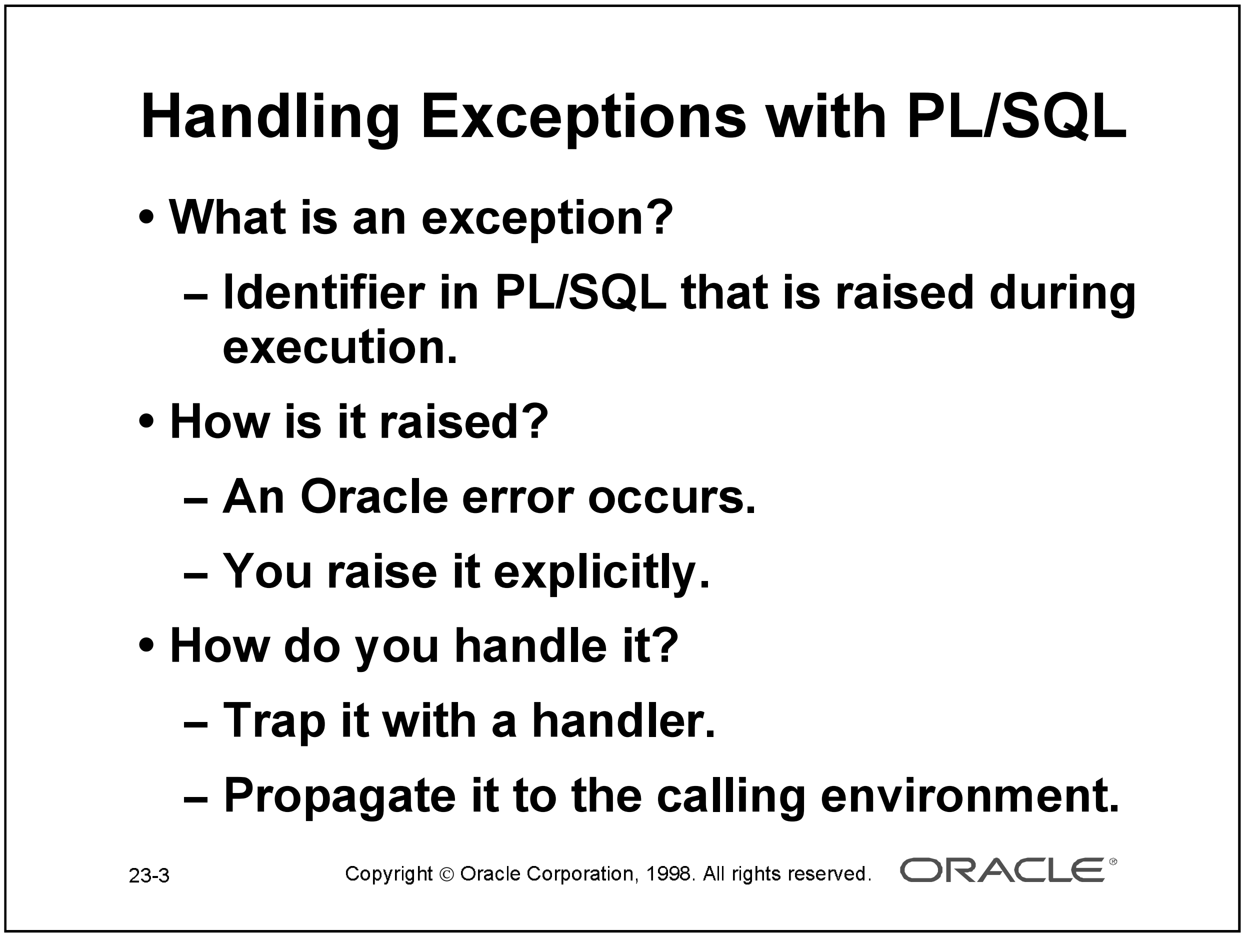
Objectives

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

- Define PL/SQL exceptions
- Recognize unhandled exceptions
- List and use different types of PL/SQL exception handlers
- Trap unanticipated errors
- Describe the effect of exception propagation in nested blocks
- Customize PL/SQL exception messages

Lesson Aim

In this lesson you will learn what PL/SQL exceptions are and how to deal with them using predefined, non-predefined, and user-defined exception handlers.



Handling Exceptions with PL/SQL

- What is an exception?
  - Identifier in PL/SQL that is raised during execution.
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

23-3

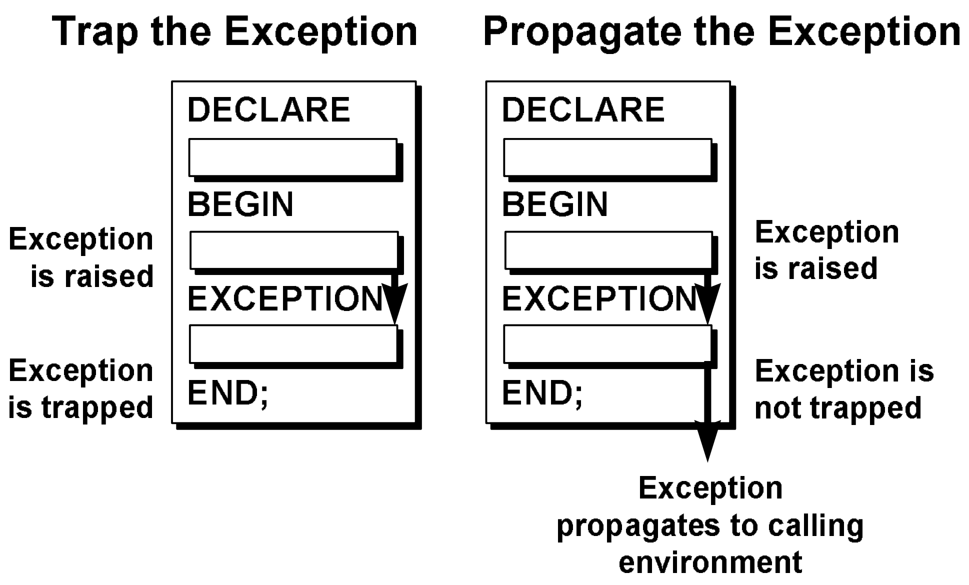
Overview

An exception is an identifier in PL/SQL, raised during the execution of a block that terminates its main body of actions. A block always terminates when PL/SQL raises an exception, but you specify an exception handler to perform final actions.

Two Methods for Raising an Exception

- An Oracle error occurs and the associated exception is raised automatically. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a select statement, then PL/SQL raises the exception NO_DATA_FOUND.
- You raise an exception explicitly by issuing the RAISE statement within the block. The exception being raised may be either user defined or predefined.

Handling Exceptions



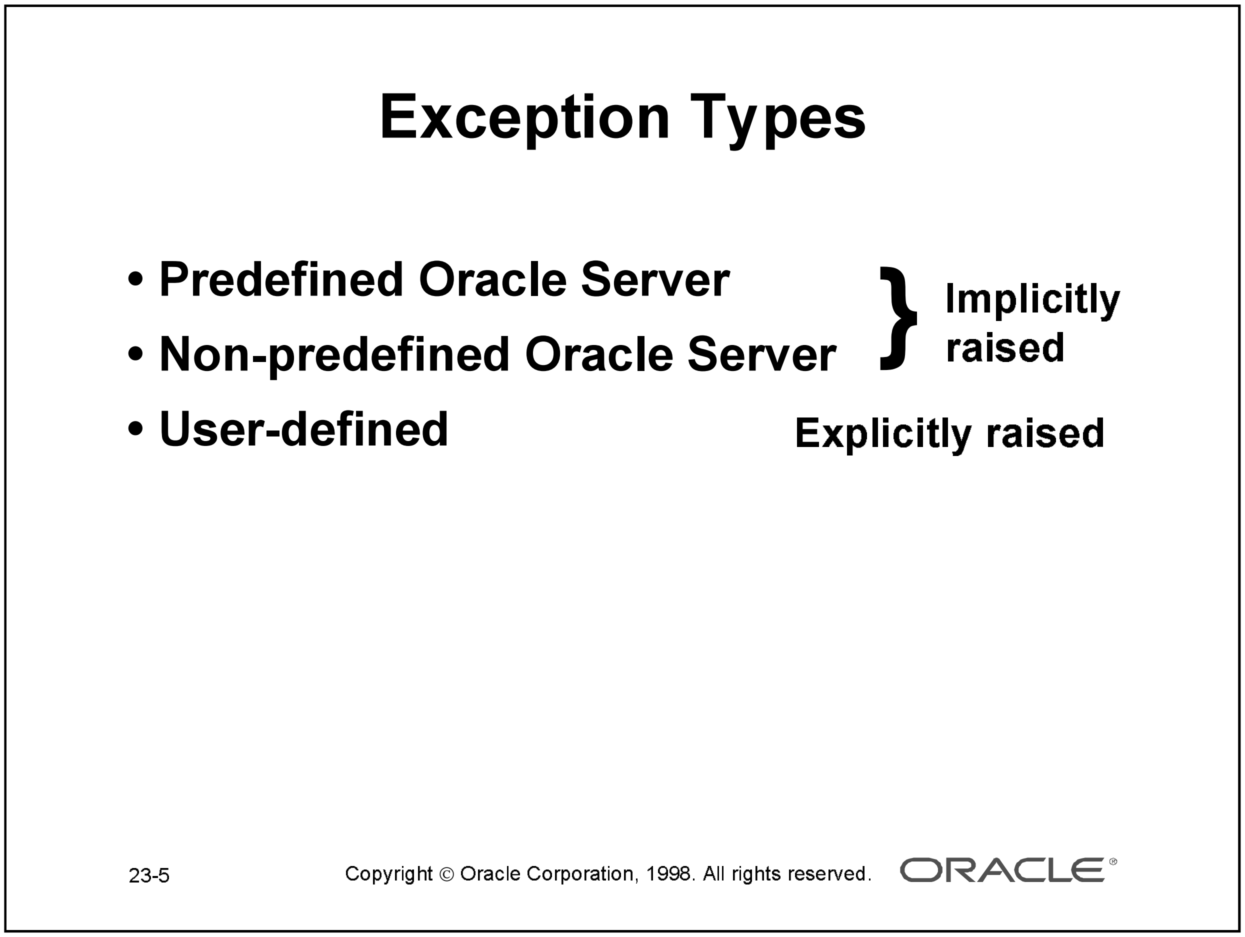
23-4

Trapping an Exception

If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or environment. The PL/SQL block terminates successfully.

Propagating an Exception

You can handle an exception by propagating it to the calling environment. If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure.



Exception Types

- Predefined Oracle Server
- Non-predefined Oracle Server
- User-defined

Implicitly raised

Explicitly raised

23-5

Exception Types

You can program for exceptions to avoid disruption at runtime. There are three types of exceptions.

Exception	Description	Directions for Handling
Predefined Oracle Server error	One of approximately 20 errors that occur most often in PL/SQL code.	Do not declare, and allow the Oracle Server to raise them implicitly.
Non-predefined Oracle Server error	Any other standard Oracle Server error.	Declare within the declarative section, and allow the Oracle Server to raise them implicitly.
User-defined error	A condition that the developer determines is abnormal.	Declare within the declarative section, *and* raise explicitly.

Note: Some application tools with client-side PL/SQL, such as Developer/2000 Forms, have their own exceptions.

Trapping Exceptions

Syntax

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

Trapping Exceptions

You can trap any error by including a corresponding routine within the exception handling section of the PL/SQL block. Each handler consists of a WHERE clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised.

In the syntax,

exception	is the standard name of a predefined exception or the name of a user-defined exception declared within the declarative section.
statement	is one or more PL/SQL or SQL statements.
OTHERS	is an optional exception-handling clause that traps unspecified exceptions.

WHEN OTHERS Exception Handler

The exception-handling section traps only those exceptions specified; any other exceptions are not trapped unless you use the OTHERS exception handler. This traps any exception not yet handled. For this reason, OTHERS is the last exception handler defined.

The OTHERS handler traps *all* exceptions not already trapped. Some Oracle tools have their own predefined exceptions that you can raise to cause events in the application. OTHERS also traps these exceptions.

Trapping Exceptions Guidelines

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

23-7

Guidelines

- Begin the exception-handling section of the block with the keyword EXCEPTION.
- Define several exception handlers, each with its own set of actions, for the block.
- When an exception occurs, PL/SQL processes *only one* handler before leaving the block.
- Place the OTHERS clause after all other exception-handling clauses.
- You can have at most one OTHERS clause.
- Exceptions cannot appear in assignment statements or SQL statements.

Trapping Predefined Oracle Server Errors

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

23-8

Trapping Predefined Oracle Server Errors

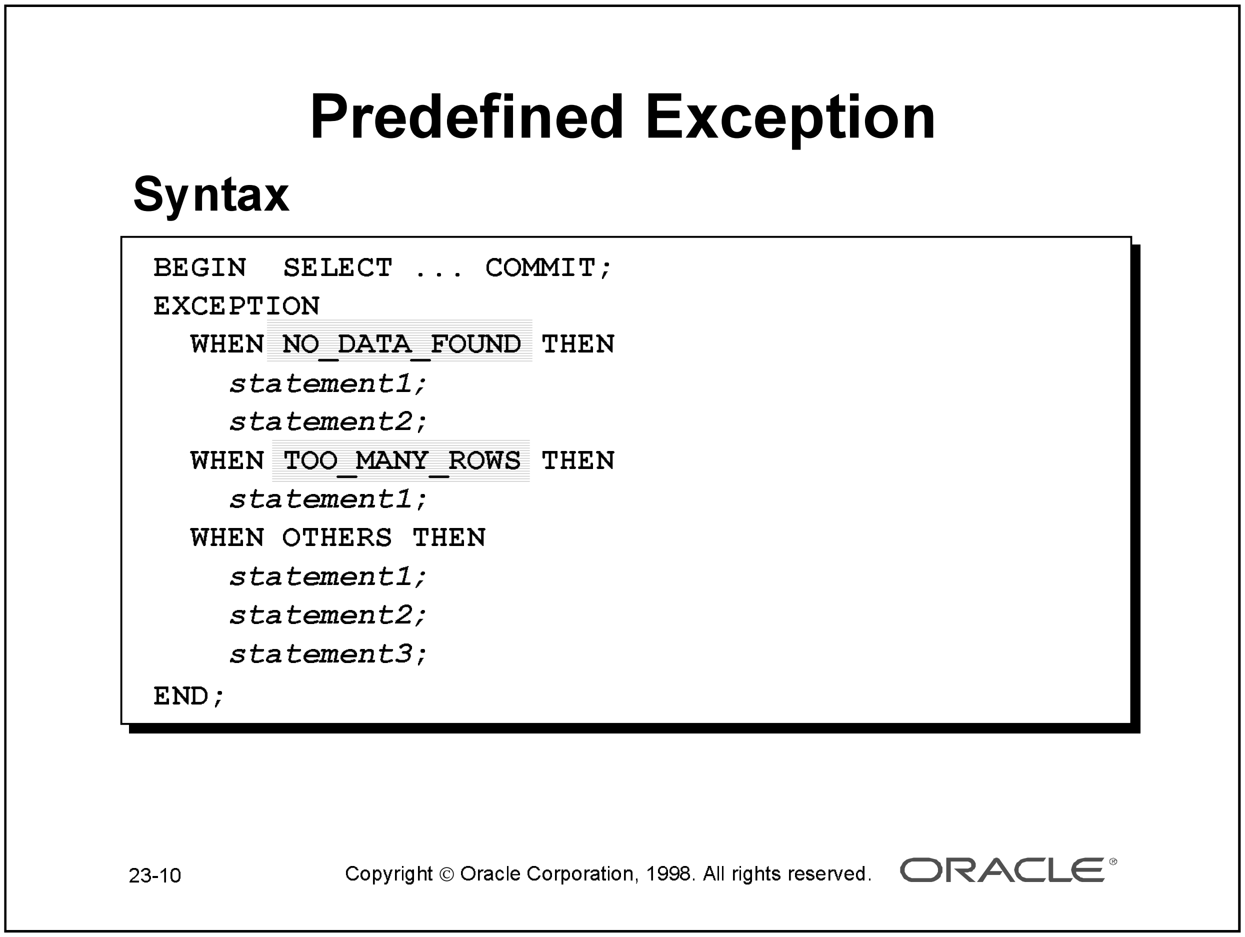
Trap a predefined Oracle Server error by referencing its standard name within the corresponding exception-handling routine.

For a complete list of predefined exceptions, see *PL/SQL User's Guide and Reference, Release 8,* "Error Handling."

Note: PL/SQL declares predefined exceptions in the STANDARD package.
It is a good idea to always consider the NO_DATA_FOUND and TOO_MANY_ROWS exceptions, which are the most common.

Predefined Exceptions

Exception Name	Oracle Server Error Number	Description
ACCESS_INTO_NULL	ORA-06530	Try to assign values to the attributes of an uninitialized object.
COLLECTION_IS_NULL	ORA-06531	Try to apply collection methods other than EXISTS to an uninitialized nested table or varray.
CURSOR_ALREADY_OPEN	ORA-06511	Try to open an already open cursor.
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value.
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred.
INVALID_NUMBER	ORA-01722	Convertion of character string to number fails.
LOGIN_DENIED	ORA-01017	Logging on to Oracle with an invalid username and/or password.
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data.
NOT_LOGGED_ON	ORA-01012	PL/SQL program issues a database call without being connected to Oracle.
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	Host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types.
STORAGE_ERROR	ORA-06500	PL/SQL runs out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Reference a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Reference a nested table or varray element using an index number that is outside the legal range (-1 for example).
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single row SELECT returned more than one row.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero.



Predefined Exception

Syntax

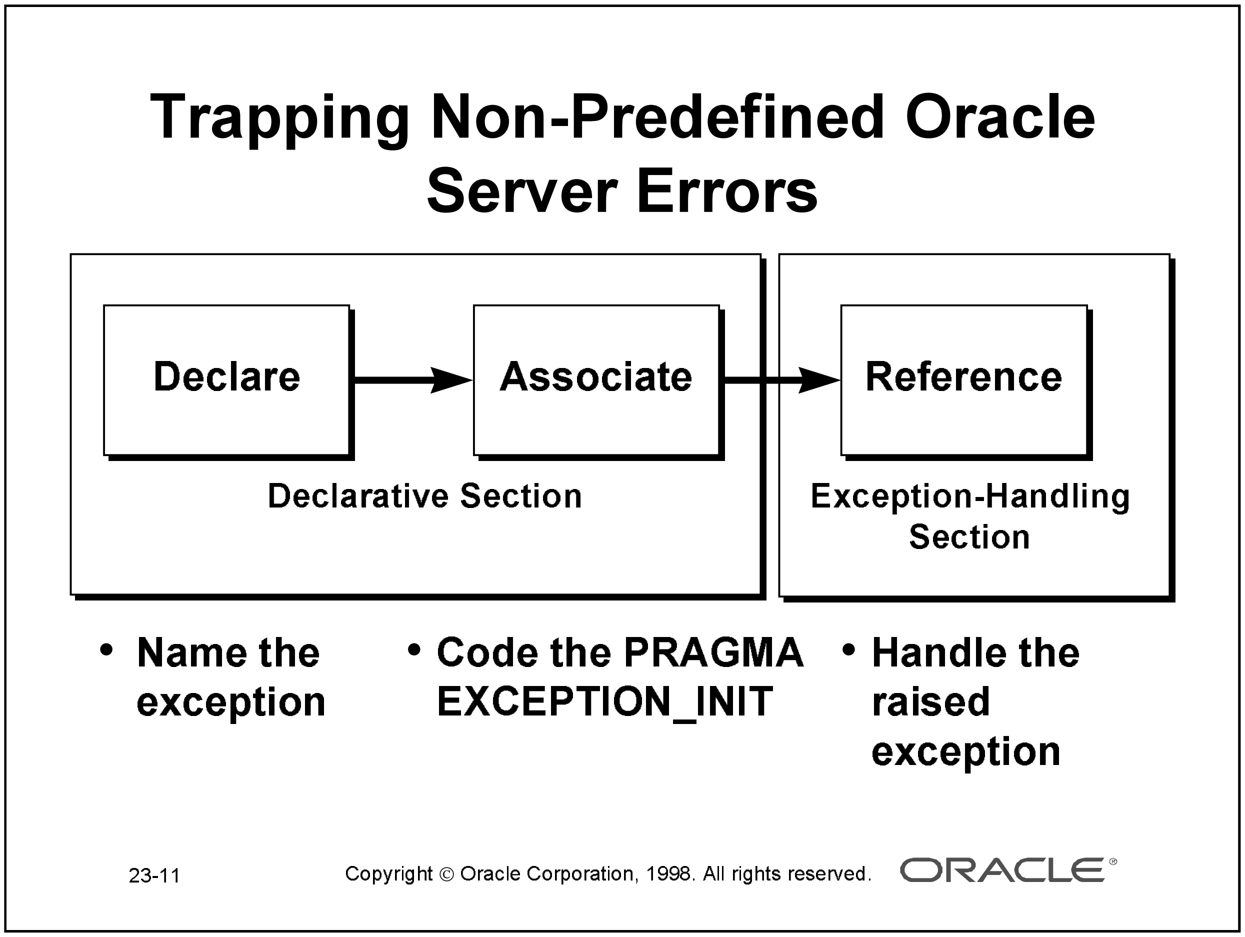
```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

23-10

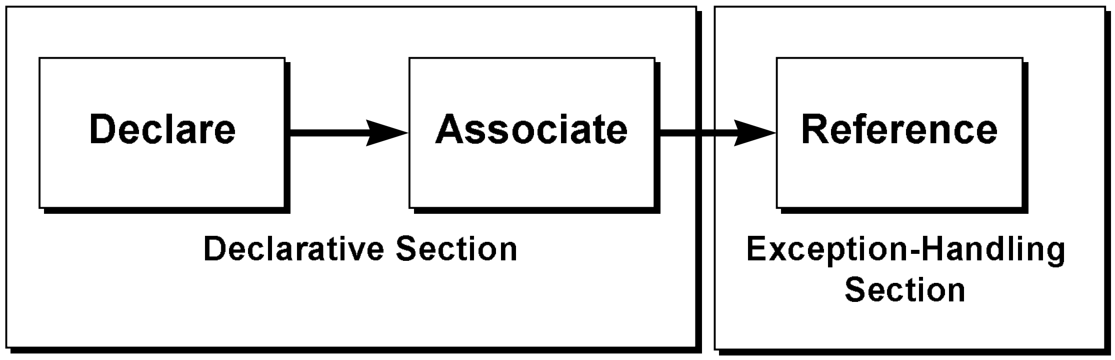
Trapping Predefined Oracle Server Exceptions

In the slide example for each exception, a message is printed out to the user.

Only one exception is raised and handled at any time.



Trapping Non-Predefined Oracle Server Errors



- Name the exception
- Code the PRAGMA EXCEPTION_INIT
- Handle the raised exception

23-11

Trapping Non-Predefined Oracle Server Errors

You trap a non-predefined Oracle Server error by declaring it first, or by using the OTHERS handler. The declared exception is raised implicitly. In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That allows you to refer to any internal exception by name and to write a specific handler for it.

Note: PRAGMA (also called pseudoinstructions) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the PL/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle Server error number.

Non-Predefined Error

Trap for Oracle Server error number -2292 an integrity constraint violation

```
DECLARE
  e_products_invalid    EXCEPTION;
  PRAGMA EXCEPTION_INIT (
             e_products_invalid, -2292);
  v_message VARCHAR2(50);
BEGIN
. . .
EXCEPTION
  WHEN e_products_invalid THEN
    :g_message := 'Product code
             specified is not valid.';
. . .
END;
```

1

2

3

Trapping a Non-Predefined Oracle Server Exception

1. Declare the name for the exception within the declarative section.

   Syntax

   ```
   exception  EXCEPTION;
   ```

where:	*exception*	is the name of the exception.

2. Associate the declared exception with the standard Oracle Server error number using the PRAGMA EXCEPTION_INIT statement.

   Syntax

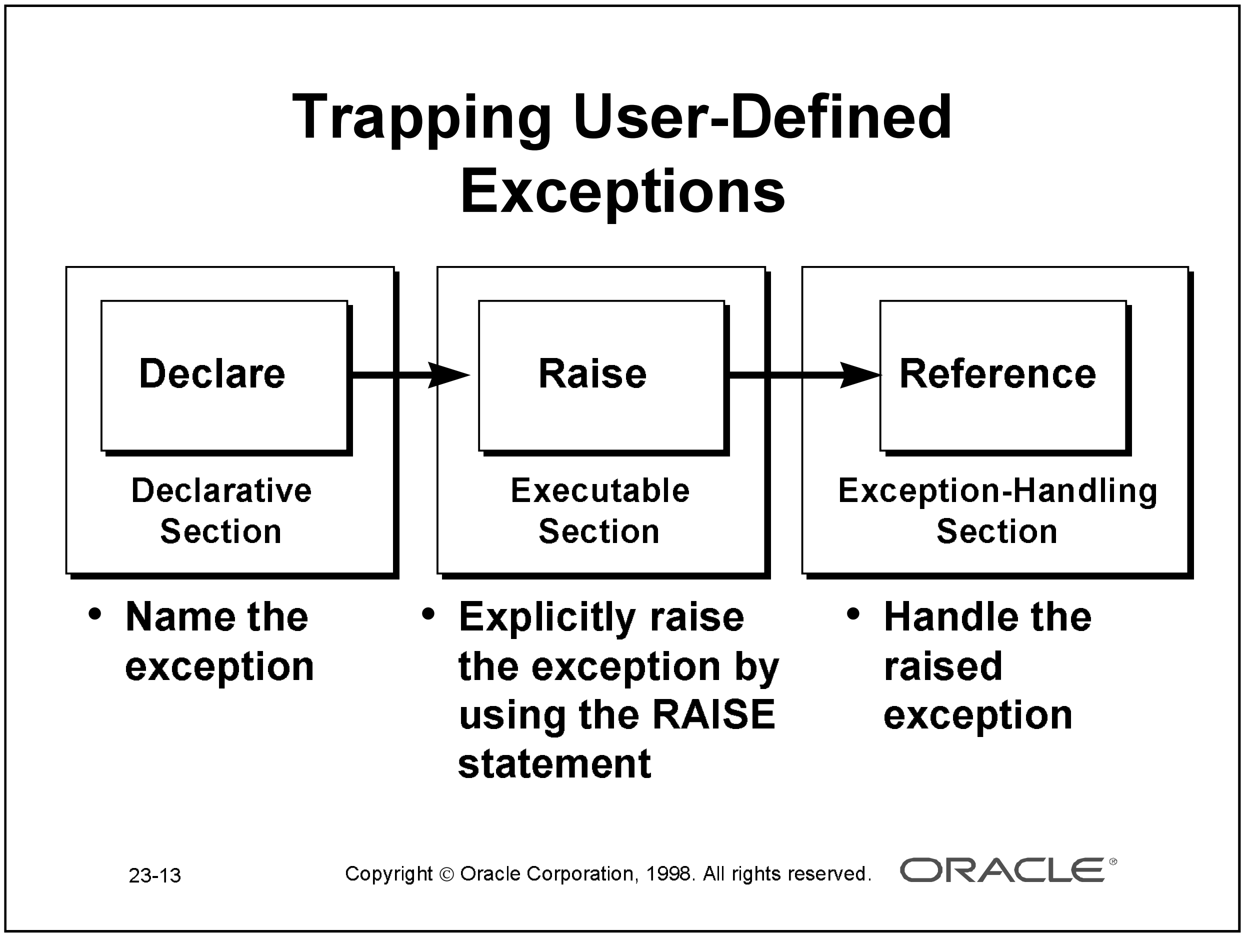
   ```
   PRAGMA EXCEPTION_INIT(exception, error_number);
   ```

where:	*exception*	is the previously declared exception.
	error_number	is a standard Oracle Server error number.

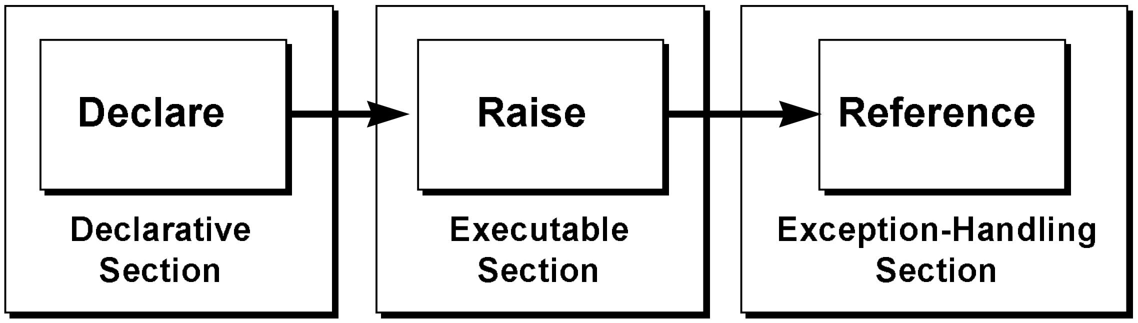
3. Reference the declared exception within the corresponding exception-handling routine.

   In the slide example: If there is product in stock, halt processing and print a message to the user.

For more information, see
Oracle Server Messages, Release 8.



Trapping User-Defined Exceptions



- Name the exception
- Explicitly raise the exception by using the RAISE statement
- Handle the raised exception

23-13

Trapping User-Defined Exceptions

PL/SQL lets you define your own exceptions. User-defined PL/SQL exceptions must be:

- Declared in the declare section of a PL/SQL block
- Raised explicitly with RAISE statements

User-Defined Exception

Example

```
[DECLARE]
  e_amount_remaining EXCEPTION;                        1
. . .
BEGIN
. . .
  RAISE e_amount_remaining;                            2
. . .
EXCEPTION
  WHEN e_amount_remaining THEN                         3
    :g_message := 'There is still an amount
                   in stock.';
. . .
END;
```

Trapping User-Defined Exceptions

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name for the user-defined exception within the declarative section.

   Syntax

   ```
   exception EXCEPTION;
   ```

   where: *exception* is the name of the exception.

2. Use the RAISE statement to raise the exception explicitly within the executable section.

   Syntax

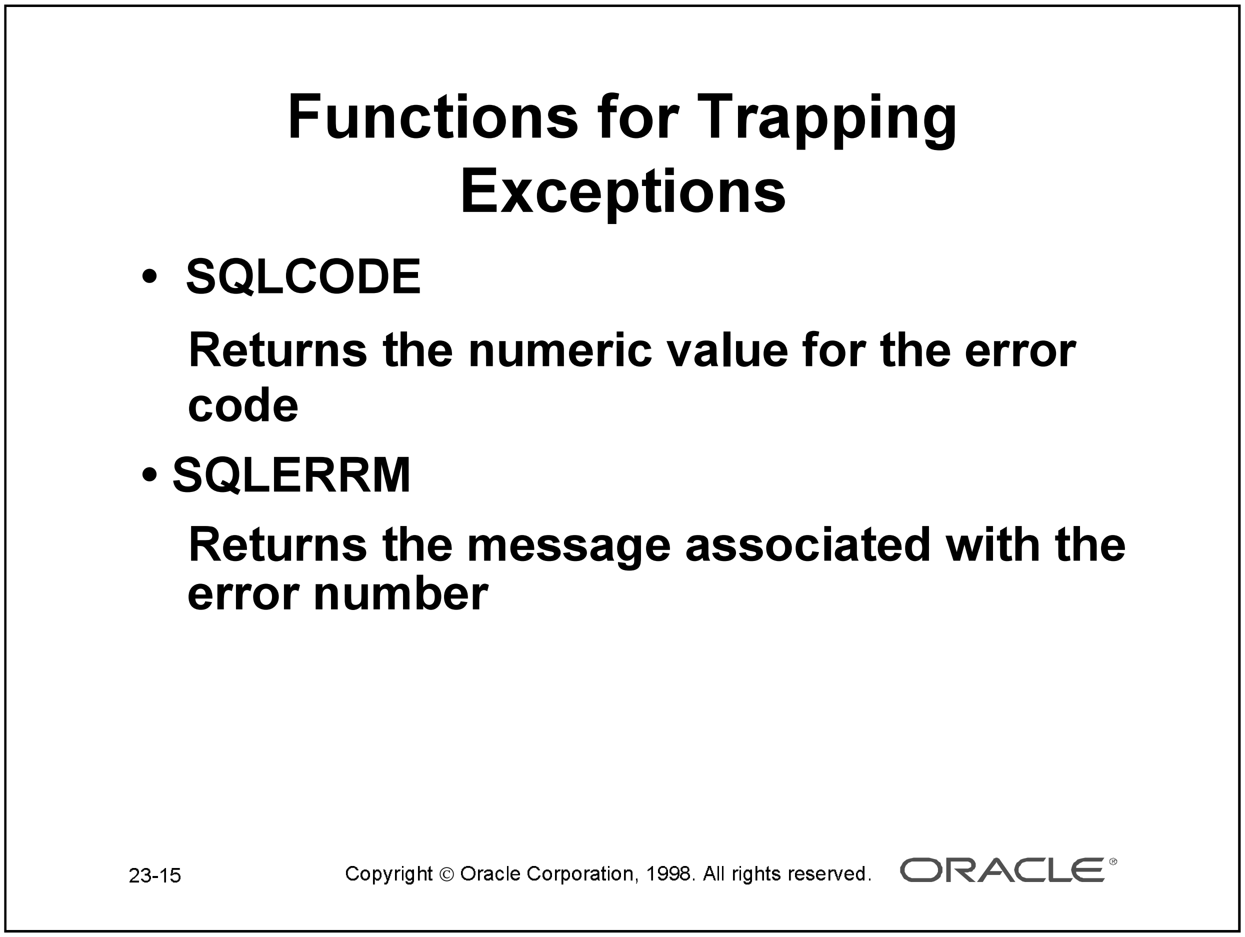
   ```
   RAISE exception;
   ```

   where: *exception* is the previously declared exception.

3. Reference the declared exception within the corresponding exception handling routine.

In the slide example: This customer has a business rule that states a that a product can not be removed from its database if there is any inventory left in-stock for this product. As there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a DELETE on the PRODUCT table, the block queries the INVENTORY table to see if there is any stock for the product in question. If so, raise an exception.

Note: Use the RAISE statement by itself within an exception handler to raise the same exception back to the calling environment.



Functions for Trapping Exceptions

- **SQLCODE**

  Returns the numeric value for the error code

- **SQLERRM**

  Returns the message associated with the error number

23-15

Error Trapping Functions

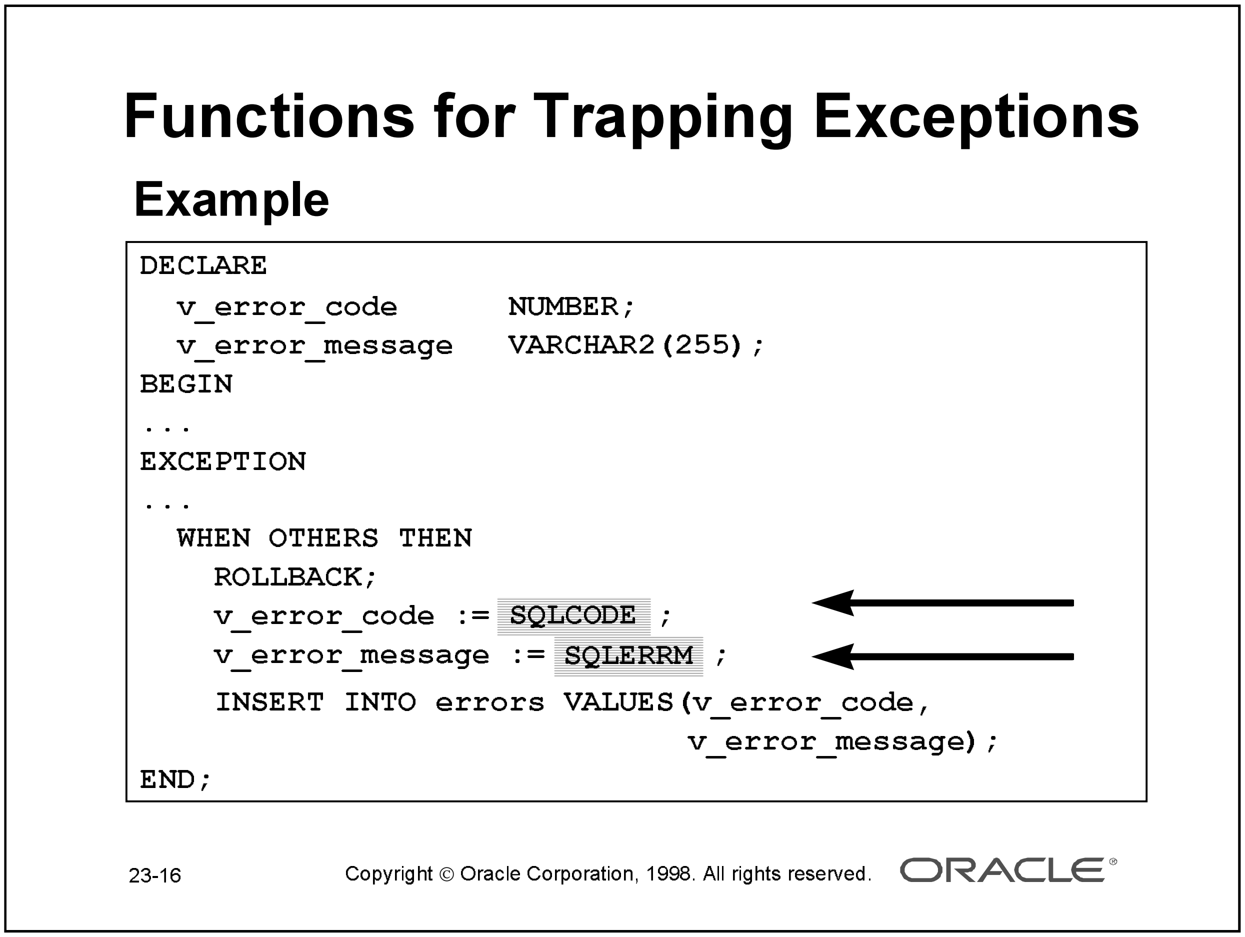
When an exception occurs, you can identify the associated error code or error message by using two functions. Based on the values of the code or message, you can decide what subsequent action to take based on the error.

SQLCODE returns the number of the Oracle error for internal exceptions. You can pass an error number to SQLERRM, which then returns the message associated with the error number.

Function	**Description**
SQLCODE	Returns the numeric value for the error code (You can assign it to a NUMBER variable.)
SQLERRM	Returns character data containing the message associated with the error number

Example SQLCODE Values

SQLCODE Value	**Description**
0	No exception encountered
1	User-defined exception
+100	NO_DATA_FOUND exception
negative number	Another Oracle Server error number



Functions for Trapping Exceptions

Example

```
DECLARE
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;             <--------
    v_error_message := SQLERRM ;          <--------
    INSERT INTO errors VALUES(v_error_code,
                              v_error_message);
END;
```

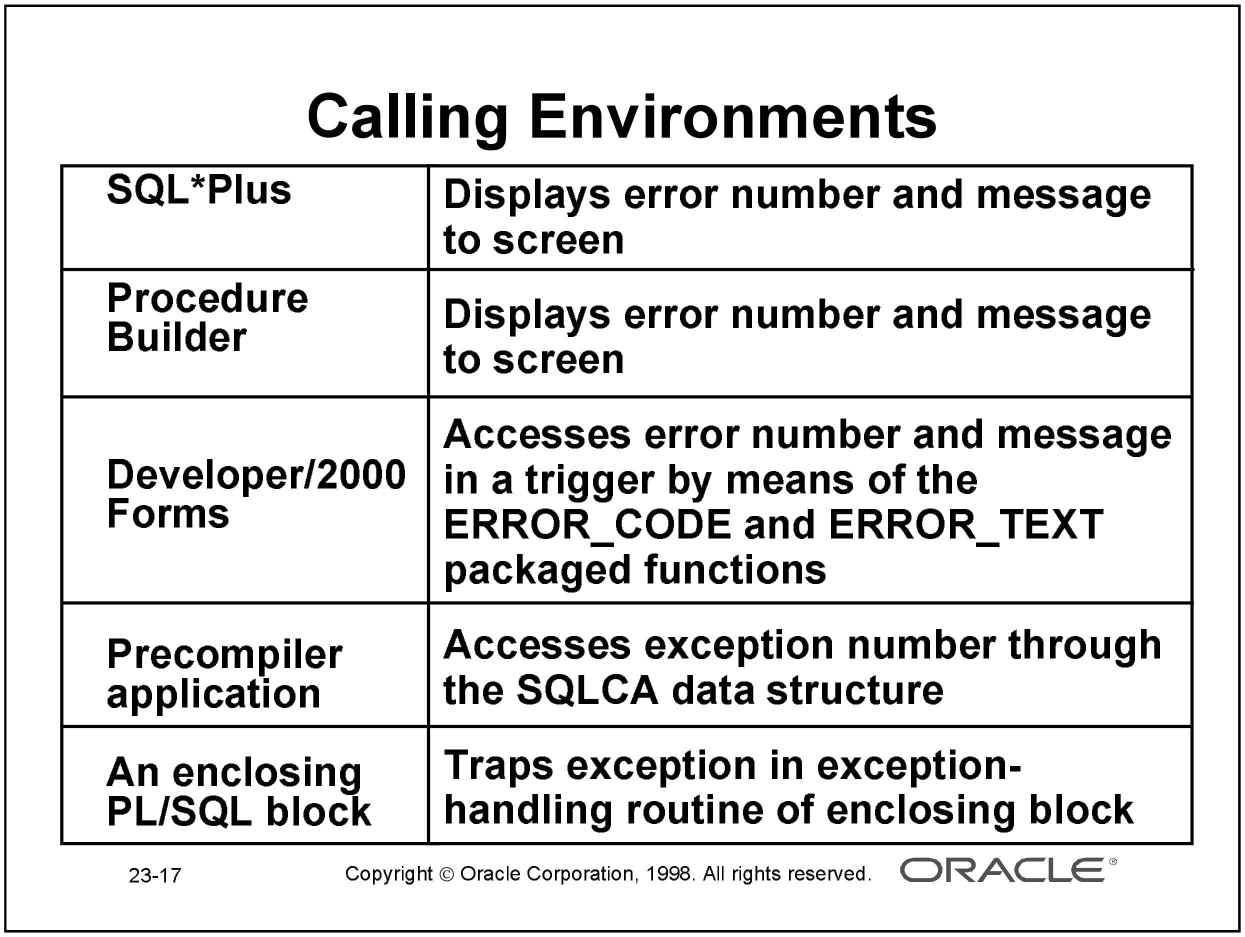
23-16 ORACLE®

Error Trapping Functions

When an exception is trapped in the WHEN OTHERS section, you can use a set of generic functions to identify those errors.

The example in the slide illustrates the values of SQLCODE and SQLERRM being assigned to variables and then those variables being used in a SQL statement.

Truncate the value of SQLERRM to a known length before attempting to write it to a variable.



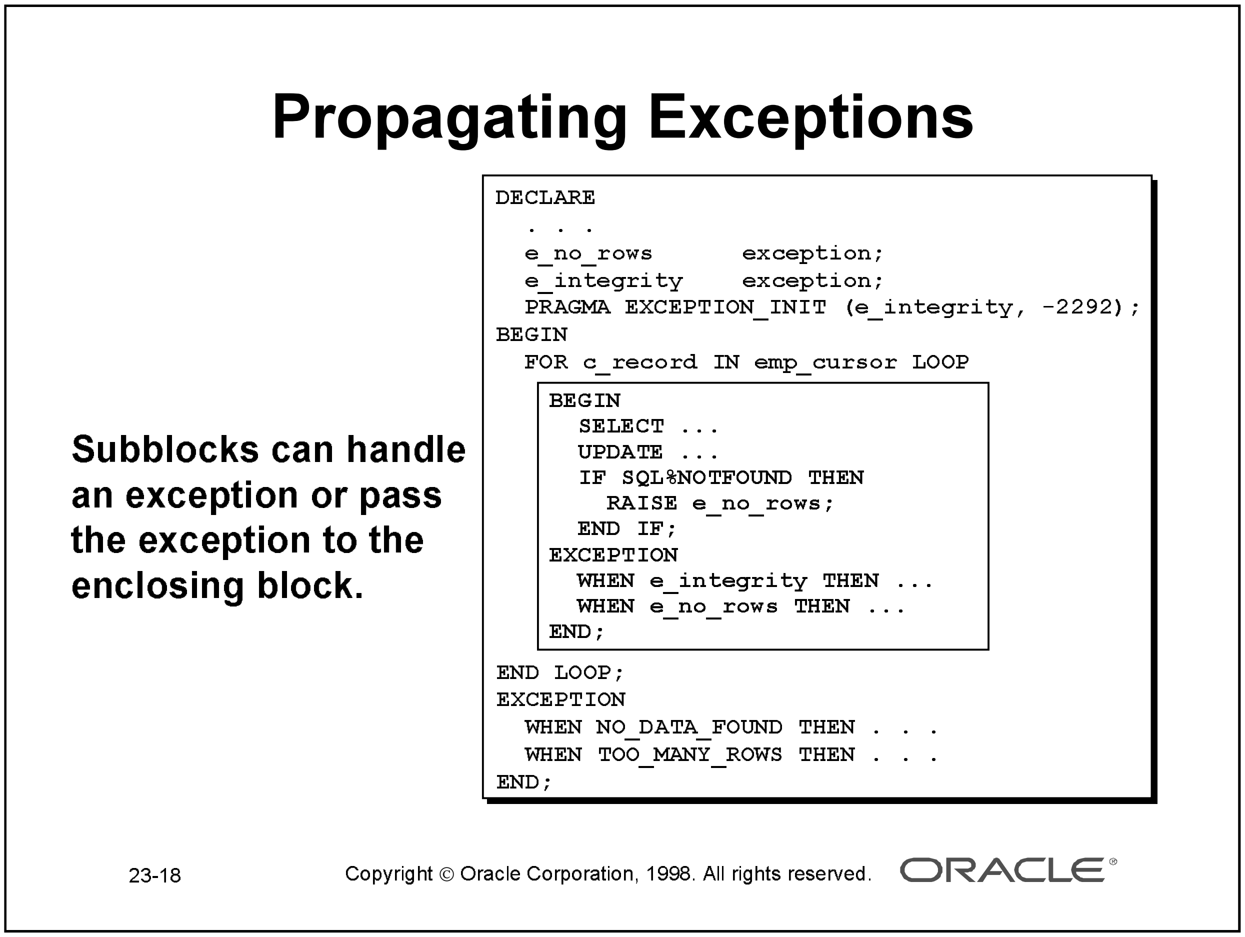
Calling Environments

SQL*Plus	Displays error number and message to screen
Procedure Builder	Displays error number and message to screen
Developer/2000 Forms	Accesses error number and message in a trigger by means of the ERROR_CODE and ERROR_TEXT packaged functions
Precompiler application	Accesses exception number through the SQLCA data structure
An enclosing PL/SQL block	Traps exception in exception-handling routine of enclosing block

23-17

Propagating Exceptions

Instead of trapping an exception within the PL/SQL block, propagate the exception to allow the calling environment to handle it. Each calling environment has its own way of displaying and accessing errors.



Propagating Exceptions

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

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

23-18

Propagating an Exception in a Subblock

When a subblock handles an exception, it terminates normally, and control resumes in the enclosing block immediately after the subblock END statement.

However, if PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates in successive enclosing blocks until it finds a handler. If none of these blocks handle the exception, an unhandled exception in the host environment results.

When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.

RAISE_APPLICATION_ERROR

Syntax

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

- **A procedure that lets you issue user-defined error messages from stored subprograms.**
- **Called only from an executing stored subprogram.**

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR you can report errors to your application and avoid returning unhandled exceptions.

In the syntax,

error_number	is a user specified number for the exception between -20000 and -20999.
message	is the user-specified message for the exception. It is a character string up to 2048 bytes long.
TRUE | FALSE	is an optional Boolean parameter. If TRUE, the error is placed on the stack of previous errors. If FALSE (the default), the error replaces all previous errors.

Example

```
...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20201,
           'Manager is not a valid employee.');
END;
```

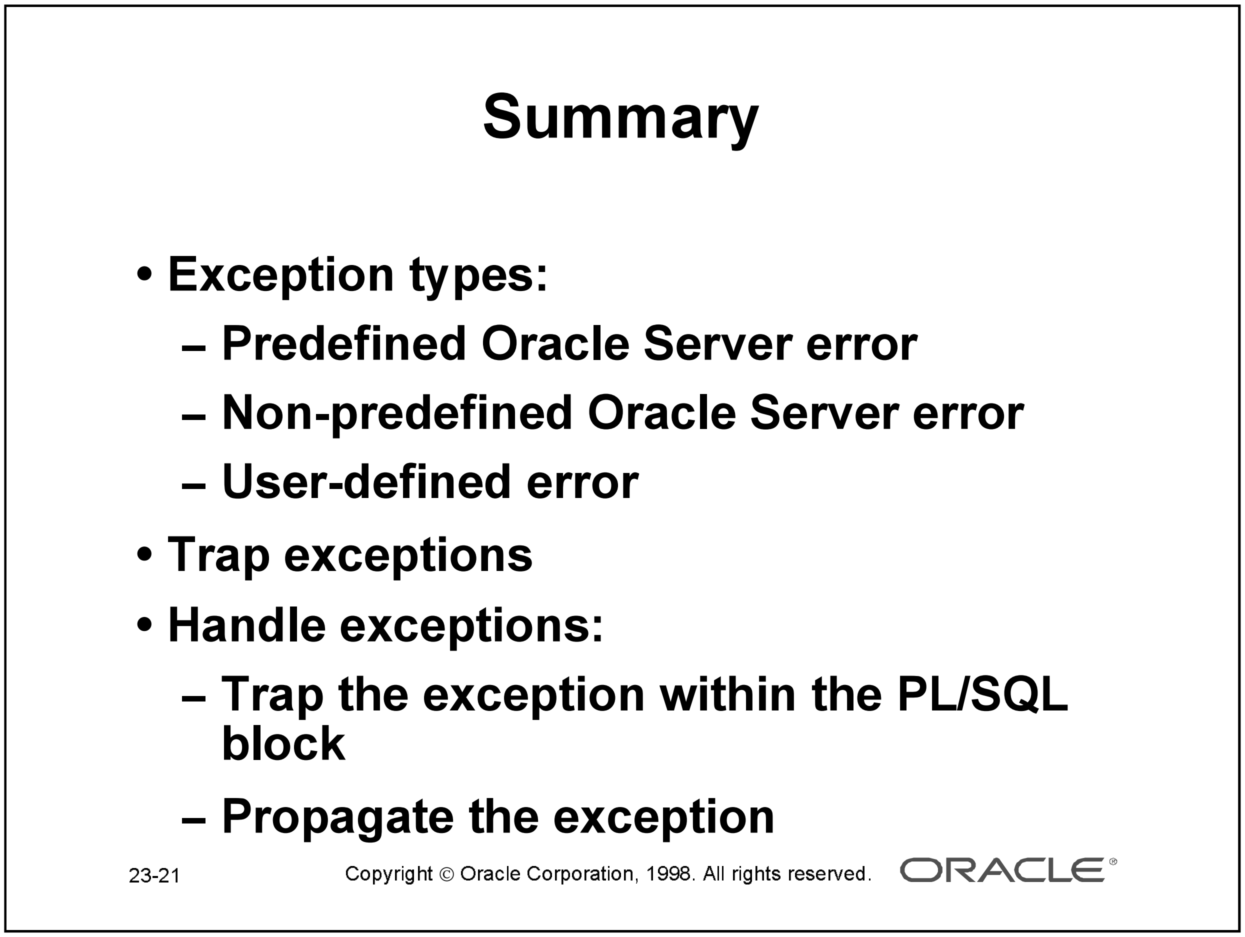
RAISE_APPLICATION_ERROR

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

23-20 ORACLE®

Example

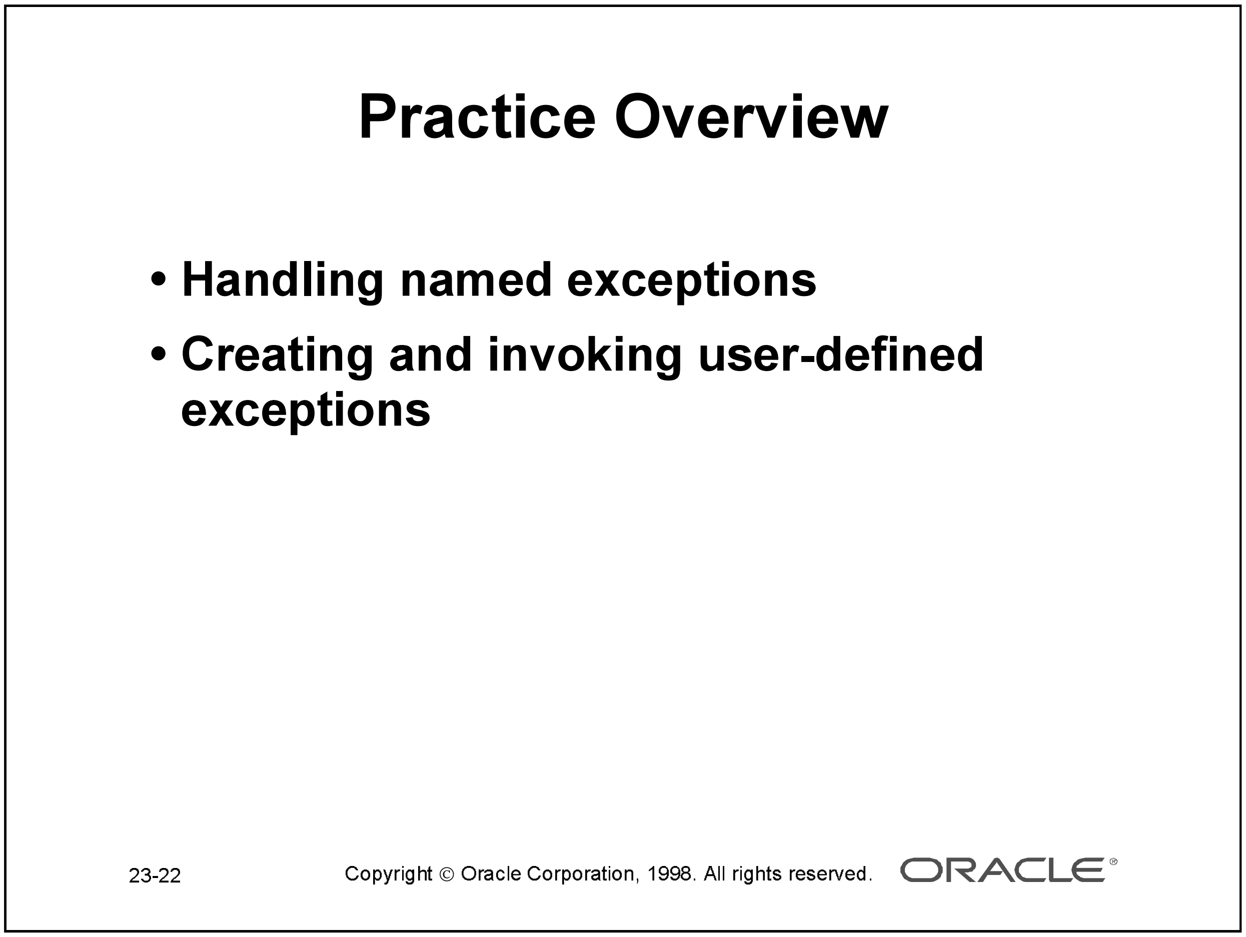
```
...
DELETE FROM emp
WHERE  mgr = v_mgr;
IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR(-20202,'This is not a valid manager');
END IF;
...
```



Summary

- Exception types:
  - Predefined Oracle Server error
  - Non-predefined Oracle Server error
  - User-defined error
- Trap exceptions
- Handle exceptions:
  - Trap the exception within the PL/SQL block
  - Propagate the exception

23-21



Practice Overview

- Handling named exceptions
- Creating and invoking user-defined exceptions

23-22

Practice Overview

In this practice, you create exception handlers for specific situations.

Practice 23

1. Write a PL/SQL block to select the name of the employee with a given salary value.
   a. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the MESSAGES table, the message "More than one employee with a salary of <*salary*>."
   b. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the MESSAGES table, the message "No employee with a salary of <*salary*>."
   c. If the salary entered returns only one row, insert into the MESSAGES table the employee's name and the salary amount.
   d. Handle any other exception with an appropriate exception handler and insert into the MESSAGES table, the message "Some other error occurred."
   e. Test the block for a variety of test cases.

```
RESULTS
--------------------------------------------
SMITH - 800
More than one employee with a salary of 3000
No employee with a salary of 6000
```

2. Modify *p18q3.sql* to add an exception handler.
   a. Write an exception handler for the error to pass a message to the user that the specified department does not exist.
   b. Execute the PL/SQL block by entering a department that does not exist.

```
Please enter the department number: 50
Please enter the department location: HOUSTON
PL/SQL procedure successfully completed.

G_MESSAGE
--------------------------------------
Department 50 is an invalid department
```

3. Write a PL/SQL block that prints the names of the employees who make plus or minus $100 of the salary value entered.
   a. If there is no employee within that salary range, print a message to the user indicating that is the case. Use an exception for this case.
   b. If there are one or more employees within that range, the message should indicate how many employees are in that salary range.
   c. Handle any other exception with an appropriate exception handler, the message should indicate that some other error occurred.

Practice 23 (continued)

```
Please enter the salary: 800
PL/SQL procedure successfully completed.

G_MESSAGE
-----------------------------------------------------------
There is 1 employee(s) with a salary between 700 and 900

Please enter the salary: 3000
PL/SQL procedure successfully completed.

G_MESSAGE
-----------------------------------------------------------
There are 3 employee(s) with a salary between 2900 and 3100

Please enter the salary: 6000
PL/SQL procedure successfully completed.

G_MESSAGE
--------------------------------------------------
There are no employee salary between 5900 and 6100
```

A

Practice Solutions

Practice 1 Solutions

1. Initiate a SQL*Plus session using the user ID and password provided by the instructor.
2. SQL*Plus commands access the database.
   False
3. The SELECT statement executes successfully?
   True

```
SQL> SELECT   ename, job, sal Salary
  2  FROM     emp;
```

4. The SELECT statement executes successfully?
   True

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

5. There are three coding errors in this statement. Can you identify them?

```
SQL> SELECT empno, ename
  2         salary x 12 ANNUAL SALARY
  3  FROM   emp;
```

- **The EMP table does not contain a column called salary. The column is called sal.**
- **The multiplication operator is *, not x, as shown in line 2.**
- **The ANNUAL SALARY alias cannot include spaces. The alias should read ANNUAL_SALARY or be enclosed in double quotation marks.**

6. Show the structure of the DEPT table. Select all data from the DEPT table.

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

7. Show the structure of the EMP table. Create a query to display the name, job, hire date and employee number for each employee, with employee number appearing first. Save your SQL statement to a file named *p1q7.sql*.

```
SQL> DESCRIBE emp
SQL> SELECT empno, ename, job, hiredate
  2  FROM   emp;
SQL> SAVE p1q7.sql
Wrote file p1q7.sql
```

8. Run your query in the file *p1q7.sql*.

```
SQL> START p1q7.sql
```

9. Create a query to display unique jobs from the EMP table.

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

If you have time, complete the following exercises.

10. Load *p1q7.sql* into the SQL buffer. Name the column headings Emp #, Employee, Job, and Hire Date, respectively. Rerun your query.

```
SQL> GET p1q7.sql
  1  SELECT empno, ename, job, hiredate
  2* FROM   emp
SQL> 1 SELECT   empno "Emp #", ename "Employee",
SQL> i
 2i  job "Job", hiredate "Hire Date"
 3i
SQL> SAVE p1q7.sql REPLACE
Wrote file p1q7.sql
SQL> START p1q7.sql
```

11. Display the name concatenated with the job, separated by a comma and space, and name the column Employee and Title.

```
SQL> SELECT      ename||', '||job "Employee and Title"
  2  FROM        emp;
```

Practice 1 Solutions (continued)

If you want extra challenge, complete the following exercises.

12. Create a query to display all the data from the EMP table. Separate each column by a comma. Name the column THE_OUTPUT.

```
SQL> SELECT empno || ',' || ename || ','|| job || ',' ||
            mgr || ',' || hiredate || ',' || sal || ',' ||
            comm || ',' || deptno THE_OUTPUT
     FROM   emp;
```

Practice 2 Solutions

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.

```
SQL> SELECT   ename, sal
  2  FROM     emp
  3  WHERE    sal > 2850;
```

```
SQL> SAVE p2q1.sql
Created file p2q1.sql;
```

2. Create a query to display the employee name and department number for employee number 47566.

```
SQL> SELECT   ename, deptno
  2  FROM     emp
  3  WHERE    empno = 7566;
```

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.

```
SQL> EDIT p2q1.sql

     SELECT   ename, sal
     FROM     emp
     WHERE    sal NOT BETWEEN 1500 AND 2850
     /

SQL> START p2q3.sql
```

4. Display the employee name, job, and start date of employees hired between February 20, 1981, and May 1, 1981. Order the query in ascending order by start date.

```
SQL> SELECT   ename, job, hiredate
  2  FROM     emp
  3  WHERE    hiredate BETWEEN '20-Feb-81' AND '01-May-81'
  4  ORDER BY hiredate;
```

Practice 2 Solutions (continued)

5. Display the employee name and department number of all employees in departments 10 and 30 in alphabetical order by name.

```
SQL> SELECT     ename, deptno
  2  FROM       emp
  3  WHERE      deptno IN (10, 30)
  4  ORDER BY   ename;
```

6. Modify *p2q3.sql* to list the name and salary of employees who earn more than $1500 and are in department 10 or 30. Label the column Employee and Monthly Salary, respectively. Resave your SQL statement to a file named *p2q6.sql*. Rerun your query.

```
SQL> EDIT p2q3.sql
     SELECT    ename "Employee", sal "Monthly Salary"
     FROM      emp
     WHERE     sal  > 1500
     AND       deptno IN (10, 30)
     /
SQL> START p2q6.sql
```

7. Display the name and hire date of every employee who was hired in 1982.

```
SQL> SELECT   ename, hiredate
  2  FROM     emp
  3  WHERE    hiredate LIKE '%82';
```

8. Display the name and title of all employees who do not have a manager.

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

9. Display the name, salary, and commission for all employees who earn commissions. Sort data in descending order of salary and commissions.

```
SQL> SELECT   ename, sal, comm
  2  FROM     emp
  3  WHERE    comm IS NOT NULL
  4  ORDER BY sal DESC, comm DESC;
```

Practice 2 Solutions (continued)

If you have time, complete the following exercises.

10. Display the names of all employees where the third letter of their name is an *A*.
    Note: There are two underscores (_) before the *A* in the WHERE clause.

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

11. Display the names of all employees that have two *Ls* in their name and are in department 30 or their manager is 7782.

```
SQL> SELECT   ename
  2  FROM     emp
  3  WHERE    ename LIKE '%L%L%'
  4  AND      deptno = 30
  5  OR       mgr = 7782;
```

If you want extra challenge, complete the following exercises.

12. Display the name, job, and salary for all employees whose job is Clerk or Analyst and their salary is not equal to $1000, $3000, or $5000.

```
SQL> SELECT   ename, job, sal
  2  FROM     emp
  3  WHERE    job IN ('CLERK', 'ANALYST')
  4  AND      sal NOT IN (1000, 3000, 5000);
```

13. Modify *p2q6.sql* to display the name, salary, and commission for all employees whose commission amount is greater than their salary increased by 10%. Rerun your query. Resave your query as *p2q13.sql.*

```
SQL> EDIT p2q6.sql
     SELECT   ename "Employee", sal "Monthly Salary", comm
     FROM     emp
     WHERE    comm  > sal * 1.1
     /
SQL> START p2q13.sql
```

Practice 3 Solutions

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

```
SQL> SELECT sysdate "Date"
  2  FROM   dual;
```

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*.

```
SQL> SELECT empno, ename, sal,
  2               ROUND(sal * 1.15, 0) "New Salary"
  3  FROM   emp;


SQL> SAVE p3q2.sql
Created file p3q2.sql;
```

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

```
SQL> START p3q2.sql
```

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.

```
SQL> EDIT p3q2.sql
     SELECT  empno, ename, sal,
             ROUND(sal * 1.15, 0) "New Salary",
             ROUND(sal * 1.15, 0) - sal "Increase"
     FROM    emp
     /
 SQL> START p3q2.sql
```

5. Display the employee's name, hire date, and salary review date, which is the first Monday after six months of service. Label the column REVIEW. Format the dates to appear in the format similar to "Sunday, the Seventh of September, 1981".

```
SQL> SELECT ename, hiredate,
  2         TO_CHAR(NEXT_DAY(ADD_MONTHS(hiredate, 6),
  3         'MONDAY'),
  4         'fmDay, "the" Ddspth "of" Month, YYYY')REVIEW
  5  FROM   emp;
```

Practice 3 Solutions (continued)

6. For each employee display the employee name and calculate the number of months between today and the date the employee was hired. Label the column MONTHS_WORKED. Order your results by the number of months employed. Round the number of months up to the closest whole number.

```
SQL> SELECT      ename, ROUND(MONTHS_BETWEEN
  2              (SYSDATE, hiredate)) MONTHS_WORKED
  3  FROM        emp
  4  ORDER BY    MONTHS_BETWEEN(SYSDATE, hiredate);
```

7. Write a query that produces the following for each employee:
<employee name> earns <salary> monthly but wants <3 times salary>. Label the column Dream Salaries.

```
SQL> SELECT  ename || ' earns '
  2          || TO_CHAR(sal, 'fm$99,999.00')
  3          || ' monthly but wants '
  4          || TO_CHAR(sal * 3, 'fm$99,999.00')
  5          || '.' "Dream Salaries"
  6  FROM    emp;
```

If you have time, complete the following exercises.

8. Create a query to display name and salary for all employees. Format the salary to be 15 characters long, left-padded with *$*. Label the column SALARY.

```
SQL> SELECT  ename,
  2          LPAD(sal, 15, '$') SALARY
  3  FROM    emp;
```

9. Write a query that will display the employee's name with the first letter capitalized and all other letters lowercase and the length of their name, for all employees whose name starts with *J*, *A*, or *M*. Give each column an appropriate label.

```
SQL> SELECT  INITCAP(ename) "Name",
  2          LENGTH(ename) "Length"
  3  FROM    emp
  4  WHERE   ename LIKE 'J%'
  5  OR      ename LIKE 'M%'
  6  OR      ename LIKE 'A%';
```

Practice 3 Solutions (continued)

10. Display the name, hire date, and day of the week on which the employee started. Label the column DAY. Order the results by the day of the week starting with Monday.

```
SQL> SELECT   ename, hiredate,
  2           TO_CHAR(hiredate, 'DAY') DAY
  3  FROM     emp
  4  ORDER BY TO_CHAR(hiredate - 1, 'd');
```

If you want extra challenge, complete the following exercises.

11. Create a query that will display the employee name and commission amount. If the employee does not earn commission, put “No Commission”. Label the column comm.

```
SQL> SELECT   ename,
  2           NVL(TO_CHAR(comm), 'No Commission') COMM
  3  FROM     emp;
```

Practice 4 Solutions

1. Write a query to display the name, department number, and department name for all employees.

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

2. Create a unique listing of all jobs that are in department 30.

```
SQL> SELECT   DISTINCT e.job, d.loc
  2  FROM     emp e, dept d
  3  WHERE    e.deptno = d.deptno
  4  AND      e.deptno = 30;
```

3. Write a query to display the employee name, department name, and location of all employees who earn a commission.

```
SQL> SELECT   e.ename, d.dname, d.loc
  2  FROM     emp e, dept d
  3  WHERE    e.deptno = d.deptno
  4  AND      e.comm IS NOT NULL;
```

4. Display the employee name and department name for all employees who have an *A* in their name. Save your SQL statement in a file called *p4q4.sql*.

```
SQL> SELECT   e.ename, d.dname
  2  FROM     emp e, dept d
  3  WHERE    e.deptno = d.deptno
  4  AND      e.ename LIKE '%A%';
```

5. Write a query to display the name, job, department number, and department name for all employees who work in DALLAS.

```
SQL> SELECT   e.ename, e.job, e.deptno, d.dname
  2  FROM     emp e, dept d
  3  WHERE    e.deptno = d.deptno
  4  AND      d.loc = 'DALLAS';
```

Practice 4 Solutions (continued)

6. Display the employee name and employee number along with their manager's name and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, respectively. Save your SQL statement to a file called *p4q6.sql*.

```
SQL> SELECT   e.ename "Employee", e.empno "Emp#",
  2           m.ename "Manager", m.empno "Mgr#"
  3   FROM    emp e, emp m
  4   WHERE   e.mgr = m.empno;
SQL> SAVE p4q6.sql
Created file p4q6.sql
```

7. Modify *p4q6.sql* to display all employees including King, who has no manager. Resave as *p4q7.sql*. Run *p4q7.sql*.

```
SQL> EDIT p4q6.sql
     SELECT   e.ename "Employee", e.empno "Emp#",
              m.ename "Manager", m.empno "Mgr#"
     FROM     emp  e, emp m
     WHERE    e.mgr = m.empno(+)
     /
SQL> START p4q7.sql
```

If you have time, complete the following exercises.

8. Create a query that will display the employee name, department number, and all the employees that work in the same department as a given employee. Give each column an appropriate label.

```
SQL> SELECT   e.deptno department, e.ename employee,
  2           c.ename colleague
  3  FROM     emp e, emp c
  4  WHERE    e.deptno = c.deptno
  5  AND      e.empno <> c.empno
  6  ORDER BY e.deptno, e.ename, c.ename;
```

Practice 4 Solutions (continued)

9. Show the structure of the SALGRADE table. Create a query that will display the name, job, department name, salary, and grade for all employees.

```
SQL> DESCRIBE salgrade

SQL> SELECT e.ename, e.job, d.dname, e.sal, s.grade
     FROM   emp e, dept d, salgrade s
     WHERE  e.deptno = d.deptno
     AND    e.sal BETWEEN s.losal AND s.hisal;
```

If you want extra challenge, complete the following exercises.

10. Create a query to display the name and hire date of any employee hired after employee Blake.

```
SQL> SELECT emp.ename, emp.hiredate
     FROM   emp, emp blake
     WHERE  blake.ename = 'BLAKE'
     AND    blake.hiredate < emp.hiredate;
```

11. Display all employees' names and hire dates along with their manager's name and hire date for all employees who were hired before their managers. Label the columns Employee, Emp Hiredate, Manager, and Mgr Hiredate, respectively.

```
SQL> SELECT e.ename "Employee", e.hiredate "Emp Hiredate",
            m.ename "Manager", m.hiredate "Mgr Hiredate"
     FROM   emp e, emp m
     WHERE  e.mgr = m.empno
     AND    e.hiredate < m.hiredate;
```

12. Create a query that displays the employees name and the amount of the salaries of the employees are indicated through asterisks. Each asterisk signifies a hundred dollars. Sort the data in descending order of salary. Label the column EMPLOYEE_AND_THEIR_SALARIES.

```
SQL> SELECT      rpad(ename, 8) ||rpad(' ', sal/100, '*')
                 EMPLOYEE_AND_THEIR_SALARIES
     FROM        emp
     ORDER BY    sal DESC;
```

Practice 5 Solutions

Determine the validity of the following statements. Circle either True or False.

1. Group functions work across many rows to produce one result.

   True

2. Group functions include nulls in calculations.

   False. Group functions ignore null values. If you want to include null values, use the NVL function.

3. The WHERE clause restricts rows prior to inclusion in a group calculation.

   True

4. Display the highest, lowest, sum, and average salary of all employees. Label the columns Maximum, Minimum, Sum, and Average, respectively. Round your results to the decimal position. Save your SQL statement in a file called *p5q4.sql.*

```
SQL> SELECT    ROUND(MAX(sal),0) "Maximum",
  2            ROUND(MIN(sal),0) "Minimum",
  3            ROUND(SUM(sal),0) "Sum",
  4            ROUND(AVG(sal),0) "Average"
  5  FROM      emp;
SQL> SAVE p5q4.sql
Created file p5q4.sql
```

5. Modify *p5q4.sql* to display the minimum, maximum, sum, and average salary for each job type. Resave to a file called *p5q5.sql*. Rerun your query.

```
SQL> EDIT p5q6.sql
     SELECT    job, ROUND(MAX(sal),0) "Maximum",
               ROUND(MIN(sal),0) "Minimum",
               ROUND(SUM(sal),0) "Sum",
               ROUND(AVG(sal),0) "Average"
     FROM      emp
     GROUP BY job
     /
SQL> START p5q5.sql
```

Practice 5 Solutions (continued)

6. Write a query to display the number of people with the same job.

```
SQL> SELECT   job, COUNT(*)
  2  FROM     emp
  3  GROUP BY job;
```

7. Determine the number of managers without listing them. Label the column Number of Managers.

```
SQL> SELECT   COUNT(DISTINCT mgr) "Number of Managers"
  2  FROM     emp;
```

8. Write a query that will display the difference between the highest and lowest salaries. Label the column DIFFERENCE.

```
SQL> SELECT   MAX(sal) - MIN(sal) DIFFERENCE
  2  FROM     emp;
```

If you have time, complete the following exercises.

9. Display the manager number and the salary of the lowest paid employee for that manager. Exclude anyone where the manager id is not known. Exclude any groups where the minimum salary is less than $1000. Sort the output in descending order of salary.

```
SQL> SELECT   mgr, MIN(sal)
  2  FROM     emp
  3  WHERE    mgr IS NOT NULL
  4  GROUP BY mgr
  5  HAVING   MIN(sal) > 1000
  6  ORDER BY  MIN(sal) DESC;
```

10. Write a query to display the department name, location name, number of employees, and the average salary for all employees in that department. Label the columns dname, loc, Number of People, and Salary, respectively.

```
SQL> SELECT   d.dname, d.loc, COUNT(*) "Number of People",
  2           ROUND(AVG(sal),2) "Salary"
  3  FROM     emp e, dept d
  4  WHERE    e.deptno = d.deptno
  5  GROUP BY d.dname, d.loc;
```

Practice 5 Solutions (continued)

If you want extra challenge, complete the following exercises.

11. Create a query that will display the total number of employees and of that total the number who were hired in 1980, 1981, 1982, and 1983. Give appropriate column headings.

```
SQL> SELECT COUNT(*) total,
  2         SUM(DECODE(TO_CHAR(hiredate, 'YYYY'),
  3                             1980,1,0))"1980",
  4         SUM(DECODE(TO_CHAR(hiredate, 'YYYY'),
  5                             1981,1,0))"1981",
  6         SUM(DECODE(TO_CHAR(hiredate, 'YYYY'),
  7                             1982,1,0))"1982",
  8         SUM(DECODE(TO_CHAR(hiredate, 'YYYY'),
  9                             1983,1,0))"1983"
 10  FROM   emp;
```

12. Create a matrix query to display the job, the salary for that job based upon department number and the total salary for that job for all departments, giving each column an appropriate heading.

```
SQL> SELECT      job "Job",
  2              SUM(DECODE(deptno, 10, sal)) "Dept 10",
  3              SUM(DECODE(deptno, 20, sal)) "Dept 20",
  4              SUM(DECODE(deptno, 30, sal)) "Dept 30",
  5              SUM(sal) "Total"
  6  FROM        emp
  7  GROUP BY    job;
```

Practice 6 Solutions

1. Write a query to display the employee name and hire date for all employees in the same department as Blake. Exclude Blake.

```
SQL> SELECT   ename, hiredate
  2  FROM     emp
  3  WHERE    deptno IN (SELECT    deptno
  4                      FROM      emp
  5                      WHERE     ename = 'BLAKE')
  6  AND      ename != 'BLAKE';
```

2. Create a query to display the employee number and name for all employees who earn more than the average salary. Sort the results in descending order of salary.

```
SQL> SELECT   empno, ename
  2  FROM     emp
  3  WHERE    sal > (SELECT AVG(sal)
  4                  FROM emp)
  5  ORDER BY sal DESC;
```

3. Write a query that will display the employee number and name for all employees who work in a department with any employee whose name contains a *T*. Save your SQL statement in a file called *p6q3.sql*.

```
SQL> SELECT   empno, ename
  2  FROM     emp
  3  WHERE    deptno IN (SELECT    deptno
  4                       FROM     emp
  5                       WHERE    ename LIKE '%T%');
SQL> SAVE p6q3.sql
Created file p6q3.sql
```

4. Display the employee name, department number, and job title for all employees whose department location is Dallas.

```
SQL> SELECT   ename, deptno, job
  2  FROM     emp
  3  WHERE    deptno IN (SELECT    deptno
  4                      FROM      dept
  5                      WHERE     loc = 'DALLAS');
```

Practice 6 Solutions (continued)

5. Display the employee name and salary of all employees who report to King.

```
SQL> SELECT ename, sal
     FROM   emp
     WHERE  mgr IN (SELECT   empno
                    FROM     emp
                    WHERE    ename = 'KING');
```

6. Display the department number, name, and job for all employees in the Sales department.

```
SQL> SELECT deptno, ename, job
     FROM   emp
     WHERE  deptno IN (SELECT  deptno
                       FROM    dept
                       WHERE   dname = 'SALES');
```

If you have time, complete the following exercises.

7. Modify *p6q3.sql* to display the employee number, name, and salary for all employees who earn more than the average salary and who work in a department with any employee with a *T* in their name. Resave as *p6q7.sql*. Rerun your query.

```
SQL> EDIT p6q3.sql
     SELECT empno, ename, sal
     FROM   emp
     WHERE  sal > (SELECT  AVG(sal)
                   FROM    emp)
     AND    deptno IN (SELECT  deptno
                       FROM    emp
                       WHERE   ename LIKE '%T%')
     /
SQL> START p6q7.sql
```

Practice 7 Solutions

1. Write a query to display the name, department number, and salary of any employee whose department number and salary matches both the department number and salary of any employee who earns a commission.

```
SQL> SELECT   ename, deptno, sal
  2  FROM     emp
  3  WHERE    (sal, deptno) IN
  4                      (SELECT    sal, deptno
  5                       FROM      emp
  6                       WHERE     comm IS NOT NULL);
```

2. Display the name, department name, and salary of any employee whose salary and commission matches both the salary and commission of any employee located in Dallas.

```
SQL> SELECT   ename, dname, sal
  2  FROM     emp e, dept d
  3  WHERE    e.deptno = d.deptno
  4  AND      (sal, NVL(comm,0)) IN
  5                      (SELECT    sal, NVL(comm,0)
  6                       FROM      emp e, dept d
  7                       WHERE     e.deptno = d.deptno
  8                       AND       d.loc = 'DALLAS');
```

3. Create a query to display the name, hire date, and salary for all employees who have both the same salary and commission as Scott.

```
SQL> SELECT   ename, hiredate, sal
  2  FROM     emp
  3  WHERE    (sal, NVL(comm,0)) IN
  4                      (SELECT    sal, NVL(comm,0)
  5                       FROM      emp
  6                       WHERE     ename = 'SCOTT')
  7  AND      ename != 'SCOTT';
```

4. Create a query to display the employees that earn a salary that is higher than the salary of any of the CLERKS. Sort the results on salary from highest to lowest.

```
SQL> SELECT   ename, job, sal
  2  FROM     emp
  3  WHERE    sal > ALL (SELECT sal
  4                      FROM   emp
  5                      WHERE  job = 'CLERK')
  6  ORDER BY sal DESC;
```

Practice 8 Solutions

Determine whether the following statements are true or false:

1. A single ampersand substitution variable prompts at most once.

   True
   However, if the variable is defined, then the single ampersand substitution variable will not prompt at all. If fact, it will pick up the value in the predefined variable.

2. The ACCEPT command is a SQL command.

   False
   The ACCEPT command is a SQL*Plus command. It is issued at the SQL prompt.

3. Write a script file to display the employee name, job, and hire date for all employees who started between a given range. Concatenate the name and job together, separated by a space and comma and label the column Employees. Prompt the user for the two ranges using the ACCEPT command. Use the format MM/DD/YY. Save the script file as *p8q3.sql*.

```
SET ECHO OFF
SET VERIFY OFF
ACCEPT low_date DATE FORMAT 'MM/DD/YY' -
PROMPT 'Please enter the low date range (''MM/DD/YY''): '
ACCEPT high_date DATE FORMAT 'MM/DD/YY' -
PROMPT 'Please enter the high date range (''MM/DD/YY''): '
COLUMN EMPLOYEES FORMAT A25
SELECT        ename ||', '|| job EMPLOYEES, hiredate
FROM          emp
WHERE         hiredate BETWEEN
                  TO_DATE('&low_date', 'MM/DD/YY')
              AND TO_DATE('&high_date', 'MM/DD/YY')
/
UNDEFINE low_date
UNDEFINE high_date
COLUMN EMPLOYEES CLEAR
SET VERIFY ON
SET ECHO ON
SQL> START p8q3.sql;
```

Practice 8 Solutions (continued)

4. Write a script to display the employee name, job, and department name for a given location. The search condition should allow for case_insensitive location searches. Save the script file as *p8q4.sql*.

```
SET ECHO OFF
SET VERIFY OFF
ACCEPT p_location PROMPT 'Please enter the location name: '
COLUMN ename HEADING "EMPLOYEE NAME" FORMAT A15
COLUMN dname HEADING "DEPARTMENT NAME" FORMAT A15
SELECT e.ename, e.job, d.dname
FROM   emp e, dept d
WHERE  e.deptno = d.deptno
AND    LOWER(d.loc) LIKE LOWER('%&p_location%')
/
UNDEFINE p_location
COLUMN ename CLEAR
COLUMN dname CLEAR
SET VERIFY ON
SET ECHO ON
SQL> START p8q4.sql
```

Practice 8 Solutions (continued)

5. Modify *p8q4.sql* to create a report containing the department name, employee name, hire date, salary and each employees annual salary for all employees in a given location. Prompt the user for the location. Label the columns DEPARTMENT NAME, EMPLOYEE NAME, START DATE, SALARY and ANNUAL SALARY, placing the labels on multiple lines. Resave the script as *p8q5.sql*.

```
  SET ECHO OFF
  SET FEEDBACK OFF
  SET VERIFY OFF
  BREAK ON  dname
  ACCEPT p_location PROMPT 'Please enter the location name: '
  COLUMN dname HEADING "DEPARTMENT|NAME" FORMAT A15
  COLUMN ename HEADING "EMPLOYEE|NAME" FORMAT A15
  COLUMN hiredate HEADING "START|DATE" FORMAT A15
  COLUMN sal HEADING "SALARY" FORMAT $99,990.00
  COLUMN asal HEADING "ANNUAL|SALARY" FORMAT $99,990.00
  SELECT    d.dname, e.ename, e.hiredate,
            e.sal, e.sal * 12 asal
  FROM      emp e, dept d
  WHERE     e.deptno = d.deptno
  AND       LOWER(d.loc) LIKE LOWER('%&p_location%')
  ORDER BY  dname
  /
  UNDEFINE p_location
  COLUMN dname CLEAR
  COLUMN ename CLEAR
  COLUMN hiredate CLEAR
  COLUMN sal CLEAR
  COLUMN asal CLEAR
  CLEAR BREAK
  SET VERIFY ON
  SET FEEDBACK ON
  SET ECHO  ON
SQL> START p8q5.sql
```

Practice 9 Solutions

Insert data into the MY_EMPLOYEE table.

1. Run the *lab9_1.sql* script to build the MY_EMPLOYEE table that will be used for the lab.

```
SQL> START lab9_1.sql
```

2. Describe the structure of the MY_EMPLOYEE table to identify the column names.

```
SQL> DESCRIBE my_employee
```

3. Add the first row of data to the MY_EMPLOYEE table from the sample data below. Do not list the columns in the INSERT clause.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	795
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750
5	Ropeburn	Audry	aropebur	1550

```
SQL> INSERT INTO my_employee
  2  VALUES (1, 'Patel', 'Ralph', 'rpatel', 795);
```

4. Populate the MY_EMPLOYEE table with the second row of sample data from the table above. This time, list the columns explicitly in the INSERT clause.

```
SQL> INSERT INTO my_employee (id, last_name, first_name,
  2                           userid, salary)
  3  VALUES (2, 'Dancs', 'Betty', 'bdancs', 860);
```

5. Confirm your addition to the table.

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

Practice 9 Solution (continued)

6. Create a script named *loademp.sql* to load rows into the MY_EMPLOYEE table interactively. Prompt the user for the employee's first name, last name, and salary. Concatenate the first letter of the first name and the first seven characters of the last name to produce the userid.

```
SET ECHO OFF
SET VERIFY OFF
ACCEPT p_first_name
PROMPT 'Please enter the employee's first name: '
ACCEPT p_last_name
PROMPT 'Please enter the employee's last name: '
ACCEPT p_id
PROMPT 'Please enter the employee number: '
ACCEPT p_salary PROMPT 'Please enter the employee's salary: '
INSERT INTO   my_employee
VALUES        (&p_id, '&p_last_name', '&p_first_name',
              substr('&p_first_name', 1, 1) ||
              substr('&p_last_name', 1, 7), &p_salary)
/
SET VERIFY ON
SET ECHO ON
```

7. Populate the table with the next two rows of sample data by running the script you created.

```
SQL> START loademp.sql
SQL> START loademp.sql
```

8. Confirm your additions to the table.

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

9. Make the data additions permanent.

```
SQL> COMMIT;
```

Practice 9 Solutions (continued)

Update and delete data in the MY_EMPLOYEE table.

10. Change the last name of employee 3 to Drexler.

```
SQL> UPDATE my_employee
  2  SET    last_name = 'Drexler'
  3  WHERE  id = 3;
```

11. Change the salary to 1000 for all employees with a salary less than 900.

```
SQL> UPDATE my_employee
  2  SET    salary = 1000
  3  WHERE  salary < 900;
```

12. Verify your changes to the table.

```
SQL> SELECT last_name, salary
  2  FROM   my_employee;
```

13. Delete Betty Dancs from the MY_EMPLOYEE table.

```
SQL> DELETE
  2  FROM   my_employee
  3  WHERE  last_name = 'Dancs'
  4  AND    first_name = 'Betty';
```

14. Confirm your changes to the table.

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

15. Commit all pending changes.

```
SQL> COMMIT;
```

Control data transaction to the MY_EMPLOYEE tables.

16. Populate the table with the last row of sample data by running the script you created in step 6.

```
SQL> START loademp.sql
```

Practice 9 Solutions (continued)

17. Confirm your addition to the table.

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

18. Mark an intermediate point in the processing of the transaction.

```
SQL> SAVEPOINT a;
```

19. Empty the entire table.

```
SQL> DELETE
  2  FROM   my_employee;
```

20. Confirm that the table is empty.

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

21. Discard the most recent DELETE operation without discarding the earlier INSERT operation.

```
SQL> ROLLBACK TO SAVEPOINT a;
```

22. Confirm that the new row is still intact.

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

23. Make the data addition permanent.

```
SQL> COMMIT;
```

Practice 10 Solutions

1. Create the DEPARTMENT table based on the table instance chart given below. Enter the syntax in a script called *p10q1.sql*, then execute the script to create the table. Confirm that the table is created.

Column Name	Id	Name
Key Type		
Nulls/Unique		
FK Table		
FK Column		
Datatype	Number	Varchar2
Length	7	25

```
SQL> EDIT p10q1.sql
     CREATE TABLE department
     (id NUMBER(7),
     name VARCHAR2(25))
     /
SQL> START p10q1.sql
SQL> DESCRIBE department
```

2. Populate the DEPARTMENT table with data from the DEPT table. Include only columns that you need.

```
SQL> INSERT INTO department
  2  SELECT      deptno, dname
  3  FROM        dept;
```

3. Create the EMPLOYEE table based on the table instance chart given below. Enter the syntax in a script called *p10q3.sql*, and then execute the script to create the table. Confirm that the table is created.

Column Name	ID	LAST_NAME	FIRST_NAME	DEPT_ID
Key Type				
Nulls/Unique				
FK Table				
FK Column				
Datatype	Number	Varchar2	Varchar2	Number
Length	7	25	25	7

```
CREATE TABLE  employee
(id           NUMBER(7),
last_name     VARCHAR2(25),
first_name    VARCHAR2(25),
dept_id       NUMBER(7))
/
SQL> START p10q3.sql
SQL> DESCRIBE employee
```

4. Modify the EMPLOYEE table to allow for longer employee last names. Confirm your modification.

```
SQL> ALTER TABLE employee
  2  MODIFY (last_name   VARCHAR2(50));
SQL> DESCRIBE employee
```

5. Confirm that both the DEPARTMENT and EMPLOYEE tables are stored in the data dictionary. (*Hint*: USER_TABLES)

```
SQL> SELECT        table_name
  2  FROM          user_tables
  3  WHERE         table_name IN ('DEPARTMENT', 'EMPLOYEE');
```

6. Create the EMPLOYEE2 table based on the structure of the EMP table, include only the EMPNO, ENAME and DEPTNO columns. Name the columns in your new table ID, LAST_NAME and DEPT_ID, respectively.

```
SQL> CREATE TABLE  employee2 AS
  2  SELECT        empno id, ename last_name, deptno dept_id
  3  FROM          emp;
```

7. Drop the EMPLOYEE table.

```
SQL> DROP TABLE    employee;
```

8. Rename the EMPLOYEE2 table to EMPLOYEE.

```
SQL> RENAME employee2 TO employee;
```

Practice 10 Solutions (continued)

9. Add a comment to the DEPARTMENT and EMPLOYEE table definitions describing the tables. Confirm your additions in the data dictionary.

```
SQL> COMMENT ON TABLE employee IS 'Employee Information';
SQL> COMMENT ON TABLE department IS 'Department Information';
SQL> COLUMN table_name FORMAT A15
SQL> COLUMN table_type FORMAT A10
SQL> COLUMN comments FORMAT A40
SQL> SELECT *
  2  FROM   user_tab_comments
  3  WHERE  table_name = 'DEPARTMENT'
  4  OR     table_name = 'EMPLOYEE';
```

Practice 11 Solutions

1. Add a table level PRIMARY KEY constraint to the EMPLOYEE table using the ID column. The constraint should be enabled at creation.

```
SQL> ALTER TABLE         employee
  2  ADD CONSTRAINT      employee_id_pk PRIMARY KEY (id);
```

2. Create a PRIMARY KEY constraint on the DEPARTMENT table using the ID column. The constraint should be enabled at creation.

```
SQL> ALTER TABLE         department
  2  ADD CONSTRAINT      department_id_pk PRIMARY KEY(id);
```

3. Add a foreign key reference on the EMPLOYEE table that will ensure that the employee is not assigned to a nonexistent department.

```
SQL> ALTER TABLE    employee
  2  ADD CONSTRAINT employee_dept_id_fk FOREIGN KEY (dept_id)
  3      REFERENCES department(id);
```

4. Confirm that the constraints were added by querying USER_CONSTRAINTS. Note the types and names of the constraints. Save your statement text in a file called *p11q4.sql*.

```
SQL> SELECT   constraint_name, constraint_type
  2  FROM     user_constraints
  3  WHERE    table_name IN ('EMPLOYEE', 'DEPARTMENT');
SQL> SAVE p11q4.sql
```

5. Display the object names and types from the USER_OBJECTS data dictionary view for EMPLOYEE and DEPARTMENT tables. You may want to format the columns for readability. Notice that the new tables and a new index were created.

```
SQL> COLUMN   object_name  FORMAT A30
SQL> COLUMN   object_type  FORMAT A30
SQL> SELECT   object_name, object_type
  2  FROM     user_objects;
  3  WHERE    object_name = 'EMPLOYEE'
  4  OR       object_name = 'DEPARTMENT';
```

If you have time, complete the following exercises.

6. Modify the EMPLOYEE table. Add a SALARY column of NUMBER data type, precision 7.

```
SQL> ALTER TABLE employee
  2  ADD (salary NUMBER(7));
```

Practice 12 Solutions

1. Create a view called EMP_VU based on the employee number, employee name, and department number from the EMP table. Change the heading for the employee name to EMPLOYEE.

```
SQL> CREATE VIEW emp_vu AS
  2  SELECT      empno, ename employee, deptno
  3  FROM        emp;
```

2. Display the content's of the EMP_VU view.

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

3. Select the view_name and text from the data dictionary USER_VIEWS.

```
SQL> COLUMN view_name FORMAT A30
SQL> COLUMN text FORMAT A50
SQL> SELECT      view_name, text
  2  FROM        user_views;
```

4. Using your view EMP_VU, enter a query to display all employee names and department numbers.

```
SQL> SELECT      employee, deptno
  2  FROM        emp_vu;
```

5. Create a view named DEPT20 that contains the employee number, employee name, and department number for all employees in department 20. Label the view column EMPLOYEE_ID, EMPLOYEE, and DEPARTMENT_ID. Do not allow an employee to be reassigned to another department through the view.

```
SQL> CREATE VIEW dept20 AS
  2  SELECT      empno employee_id, ename employee,
  3              deptno department_id
  4  FROM        emp
  5  WHERE       deptno = 20
  6  WITH CHECK OPTION CONSTRAINT emp_dept_20;
```

Practice 12 Solutions (continued)

6. Display the structure and contents of the DEPT20 view.

```
SQL> DESCRIBE dept20
SQL> SELECT *
  2  FROM   dept20;
```

7. Attempt to reassign Smith to department 30.

```
SQL> UPDATE dept20
  2  SET    department_id = 30
  3  WHERE  employee = 'SMITH';
```

If you have time, complete the following exercises.

8. Create a view called SALARY_VU based on the employee name, department name, salary and salary grade for all employees. Label the columns Employee, Department, Salary and Grade, respectively.

```
SQL> CREATE VIEW salary_vu AS
  2  SELECT ename employee, dname department,
  3         sal salary, grade
  4  FROM   emp e, dept d, salgrade s
  5  WHERE  e.deptno = d.deptno
  6  AND    e.sal between s.losal and s.hisal;
```

Practice 13 Solutions

1. Create a sequence to be used with the DEPARTMENT table's primary key column. The sequence should start at 60 and have a maximum value of 200. Have your sequence increment by ten numbers. Name the sequence DEPT_ID_SEQ.

```
SQL> CREATE SEQUENCE dept_id_seq
  2  START WITH 60
  3  INCREMENT BY 10
  4  MAXVALUE 200;
```

2. Write a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script *p13q2.sql*. Execute your script.

```
SQL> EDIT p13q2.sql
     SELECT    sequence_name, max_value,
               increment_by, last_number
     FROM      user_sequences
     /
SQL> START p13q2.sql
```

3. Write an interactive script to insert a row into the DEPARTMENT table. Name your script *p13q3.sql*. Be sure to use the sequence that you created for the ID column. Create a customized prompt to enter the department name. Execute your script. Add two departments named Education and Administration. Confirm your additions.

```
SQL> EDIT p13q3.sql
     SET ECHO OFF
     SET VERIFY OFF
     ACCEPT name PROMPT 'Please enter the department name: '
     INSERT INTO          department (id, name)
     VALUES    (dept_id_seq.NEXTVAL, '&name')
     /
     SET VERIFY ON
     SET ECHO ON
SQL> START p13q3.sql
SQL> SELECT    *
  2  FROM      department;
```

4. Create a non-unique index on the FOREIGN KEY column in the EMPLOYEE table.

```
SQL> CREATE INDEX employee_dept_id_idx ON employee (dept_id);
```

Practice 13 Solutions (continued)

5. Display the indexes and uniqueness that exist in the data dictionary for the EMPLOYEE table. Save the statement into a script named *p13q5.sql*.

```
SQL> SELECT index_name, table_name, uniqueness
  2  FROM   user_indexes
  3  WHERE  table_name = 'EMPLOYEE';
SQL> SAVE p13q5.sql
```

Practice 14 Solutions

1. What privilege should a user be given to log in to the Oracle8 Server? Is this privilege a system or object privilege?

   The CREATE SESSION system privilege.

2. What privilege should a user be given to create tables?

   The CREATE TABLE privilege.

3. If you create a table, who can pass along privileges to other users on your table?

   You can, or anyone you have given those privileges to by using the WITH GRANT OPTION.

4. You are the DBA. You are creating many users who require the same system privileges. What would you use to make your job easier?

   Create a role containing the system privileges and grant the role to the users.

5. What statement do you use to change your password?

   The ALTER USER statement.

6. Grant another user access to your DEPT table. Have the user grant you query access to his or her DEPT table.

```
Team 2 executes the GRANT statement.
SQL> GRANT     select
  2  ON        dept
  3  TO        <user1>;
Team 1 executes the GRANT statement.
SQL> GRANT     select
  2  ON        dept
  3  TO        <user2>;
WHERE user1 is the name of team 1 and user2 is the name of
     team 2.
```

7. Query all the rows in your DEPT table.

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

Practice 14 Solutions (continued)

8. Add a new row to your DEPT table. Team 1 should add Education as department number 50. Team 2 should add Administration as department number 50. Make the changes permanent.

```
Team 1 executes this INSERT statement.
SQL> INSERT INTO dept(deptno, dname)
  2  VALUES (50, 'Education');
SQL> COMMIT;
Team 2 executes this INSERT statement.
SQL> INSERT INTO dept(deptno, dname)
  2  VALUES (50, 'Administration');
SQL> COMMIT;
```

9. Create a synonym for the other team's DEPT table.

```
Team 1 creates a synonym named team2.
SQL> CREATE SYNONYM  team2
  2  FOR <user2>.DEPT;
Team 2 creates a synonym named team1.
SQL> CREATE SYNONYM  team1
  2  FOR <user1>.DEPT;
```

10. Query all the rows in the other team's DEPT table by using your synonym.

```
Team 1 executes this SELECT statement.
SQL> SELECT *
  2  FROM   team2;
Team 2 executes this SELECT statement.
SQL> SELECT *
  2  FROM   team1;
```

Practice 14 Solutions (continued)

11. Query the USER_TABLES data dictionary to see information about the tables that you own.

```
SQL> SELECT table_name
  2  FROM   user_tables;
```

12. Query the ALL_TABLES data dictionary view to see information about all the tables that you can access. Exclude tables that are owned by you.

```
SQL> SELECT table_name, owner
  2  FROM   all_tables
  3  WHERE  owner  != <your account>;
```

13. Revoke the SELECT privilege from the other team.

```
Team 1 revokes the privilege.
SQL> REVOKE select
  2  ON     dept
  3  FROM   user2;
Team 1 revokes the privilege.
SQL> REVOKE select
  2  ON     dept
  3  FROM   user1;
```

Practice 15 Solutions

1. Create the tables based on the table instance charts below. Choose the appropriate data types and be sure to add integrity constraints.

   a. Table name: MEMBER

Column _Name	MEMBER_ ID	LAST_ NAME	FIRST_ NAME	ADDRESS	CITY	PHONE	JOIN_ DATE
Key Type	PK						
Null/ Unique	NN,U	NN					NN
Default Value							System Date
Data Type	Number	Varchar2	Varchar2	Varchar2	Varchar2	Varchar2	Date
Length	10	25	25	100	30	15	

```
CREATE TABLE  member
(member_id    NUMBER(10)
              CONSTRAINT member_member_id_pk PRIMARY KEY,
last_name     VARCHAR2(25)
              CONSTRAINT member_last_name_nn NOT NULL,
first_name    VARCHAR2(25),
address       VARCHAR2(100),
city          VARCHAR2(30),
phone         VARCHAR2(15),
join_date     DATE DEFAULT SYSDATE
              CONSTRAINT member_join_date_nn NOT NULL);
```

Practice 15 Solutions (continued)

b. Table name: TITLE

Column_ Name	TITLE_ID	TITLE	DESCRIPTION	RATING	CATEGORY	RELEASE_ DATE
Key Type	PK					
Null/ Unique	NN,U	NN	NN			
Check				G, PG, R, NC17, NR	DRAMA, COMEDY, ACTION, CHILD, SCIFI, DOCUMEN TARY	
Data Type	Number	Varchar2	Varchar2	Varchar2	Varchar2	Date
Length	10	60	400	4	20	

```
CREATE TABLE title
(title_id    NUMBER(10)
   CONSTRAINT title_title_id_pk PRIMARY KEY,
 title       VARCHAR2(60)
   CONSTRAINT title_title_nn NOT NULL,
 description VARCHAR2(400)
   CONSTRAINT title_description_nn NOT NULL,
 rating      VARCHAR2(4)
   CONSTRAINT title_rating_ck CHECK
        (rating IN ('G', 'PG', 'R', 'NC17', 'NR')),
 category    VARCHAR2(20),
   CONSTRAINT title_category_ck CHECK
        (category IN ('DRAMA', 'COMEDY', 'ACTION',
        'CHILD', 'SCIFI', 'DOCUMENTARY')),
 release_dateDATE);
```

c. Table name: TITLE_COPY

Column Name	COPY_ID	TITLE_ID	STATUS
Key Type	PK	PK,FK	
Null/ Unique	NN,U	NN,U	NN
Check			AVAILABLE, DESTROYED, RENTED, RESERVED
Data Type	Number	Number	Varchar2
Length	10	10	15

```
CREATE TABLE title_copy
(copy_id       NUMBER(10),
 title_id      NUMBER(10)
CONSTRAINT title_copy_title_if_fk REFERENCES title(title_id),
 status        VARCHAR2(15)
               CONSTRAINT title_copy_status_nn NOT NULL
               CONSTRAINT title_copy_status_ck CHECK
               (status IN ('AVAILABLE', 'DESTROYED',
               'RENTED', 'RESERVED')),
CONSTRAINT title_copy_copy_id_title_id_pk
               PRIMARY KEY (copy_id, title_id));
```

Practice 15 Solutions (continued)

d. Table name: RENTAL

Column Name	BOOK_ DATE	MEMBER_ ID	COPY_ ID	ACT_RET_ DATE	EXP_RET_ DATE	TITLE_ID
Key Type	PK	PK,FK2	PK,FK			PK,FK2
Default Value	System Date				2 days after book date	
FK Ref Table		member	title_copy			title_copy
FK Ref Col		member_id	copy_id			title_id
Data Type	Date	Number	Number	Date	Date	Number
Length		10	10			10

```
CREATE TABLE rental
(book_date  DATE DEFAULT SYSDATE,
 member_id  NUMBER(10)
   CONSTRAINT rental_member_id_fk
   REFERENCES member(member_id),
copy_id          NUMBER(10),
act_ret_date     DATE,
exp_ret_date     DATE DEFAULT SYSDATE + 2,
title_id    NUMBER(10),
CONSTRAINT rental_book_date_copy_title_pk PRIMARY KEY
       (book_date, member_id, copy_id, title_id),
CONSTRAINT rental_copy_id_title_id_fk FOREIGN KEY
       (copy_id, title_id)
       REFERENCES title_copy(copy_id, title_id));
```

Practice 15 Solutions (continued)

e. Table name: RESERVATION

Column_Name	RES_DATE	MEMBER_ID	TITLE_ID
Key Type	PK	PK,FK1	PK,FK2
Null/ Unique	NN,U	NN,U	NN
FK Ref Table		MEMBER	TITLE
FK Ref Column		member_id	title_id
Data Type	Date	Number	Number
Length		10	10

```
CREATE TABLE reservation
(res_date   DATE,
 member_id  NUMBER(10)
        CONSTRAINT reservation_member_id
        REFERENCES member(member_id),
 title_id   NUMBER(10)
        CONSTRAINT reservation_title_id
        REFERENCES title(title_id),
 CONSTRAINT reservation_resdate_mem_tit_pk PRIMARY KEY
        (res_date, member_id, title_id));
```

Practice 15 Solutions (continued)

2. Verify that the tables and constraints were created properly by checking the data dictionary.

```
SQL> SELECT   table_name
  2  FROM     user_tables
  3  WHERE    table_name IN ('MEMBER', 'TITLE', 'TITLE_COPY',
  4                          'RENTAL', 'RESERVATION');
```

```
SQL> COLUMN   constraint_name FORMAT A20
SQL> COLUMN   table_name FORMAT  A15
SQL> SELECT   constraint_name, constraint_type,
  2           table_name
  3   FROM    user_constraints
  4   WHERE   table_name IN ('MEMBER', 'TITLE', 'TITLE_COPY',
  5                          'RENTAL', 'RESERVATION');
```

3. Create sequences to uniquely identify each row in the MEMBER table and the TITLE table.

   a. Member number for the MEMBER table: start with 101; do not allow caching of the values. Name the sequence member_id_seq.

```
SQL> CREATE SEQUENCE member_id_seq
  2  START WITH 101
  3  NOCACHE;
```

   b. Title number for the TITLE table: start with 92; no caching. Name the sequence title_id_seq.

```
SQL> CREATE SEQUENCE title_id_seq
  2  START WITH 92
  3  NOCACHE;
```

   c. Verify the existence of the sequences in the data dictionary.

```
SQL> SELECT   sequence_name, increment_by, last_number
  2  FROM     user_sequences
  3  WHERE    sequence_name IN ('MEMBER_ID_SEQ',
  4                             'TITLE_ID_SEQ');
```

Practice 15 Solutions (continued)

4. Add data to the tables. Create a script for each set of data to add.
   a. Add movie titles to the TITLE table. Write a script to enter the movie information Save the script as *p15q4a.sql.* Use the sequences to uniquely identify each title. Remember that single quotation marks in a character field must be specially handled. Verify your additions.

```
SQL> EDIT p15q4a.sql
SET ECHO OFF
INSERT INTO title(title_id, title, description, rating,
                  category, release_date)
VALUES (title_id_seq.NEXTVAL, 'Willie and Christmas Too',
       'All of Willie''s friends made a Christmas list for
        Santa, but Willie has yet to add his own wish list.',
       'G', 'CHILD', '05-OCT-95')
/
INSERT INTO title(title_id , title, description, rating,
                  category, release_date)
VALUES (title_id_seq.NEXTVAL, 'Alien Again', 'Yet another
       installment of science fiction history.  Can the
       heroine save the planet from the alien life form?',
       'R', 'SCIFI', '19-MAY-95')
/
INSERT INTO title(title_id, title, description, rating,
                  category, release_date)
VALUES (title_id_seq.NEXTVAL, 'The Glob', 'A meteor crashes
       near a small American town and unleashed carnivorous
       goo in this classic.', 'NR', 'SCIFI', '12-AUG-95')
/
INSERT INTO title(title_id, title, description, rating,
                  category, release_date)
VALUES (title_id_seq.NEXTVAL, 'My Day Off', 'With a little
       luck and a lot ingenuity, a teenager skips school for
       a day in New York.', 'PG', 'COMEDY', '12-JUL-95')
/
...
COMMIT
/
SET ECHO ON

SQL> SELECT   title
  2  FROM     title;
```

Practice 15 Solutions (continued)

Title	Description	Rating	Category	Release_date
Willie and Christmas Too	All of Willie's friends made a Christmas list for Santa, but Willie has yet to add his own wish list.	G	CHILD	05-OCT-95
Alien Again	Yet another installation of science fiction history. Can the heroine save the planet from the alien life form?	R	SCIFI	19-MAY-95
The Glob	A meteor crashes near a small American town and unleashed carnivorous goo in this classic.	NR	SCIFI	12-AUG-95
My Day Off	With a little luck and a lot of ingenuity, a teenager skips school for a day in New York	PG	COMEDY	12-JUL-95
Miracles on Ice	A six-year-old has doubts about Santa Claus but she discovers that miracles really do exist.	PG	DRAMA	12-SEP-95
Soda Gang	After discovering a cache of drugs, a young couple find themselves pitted against a vicious gang.	NR	ACTION	01-JUN-95

Practice 15 Solutions (continued)

b. Add data to the MEMBER table. Write a script named *p15q4b.sql* to prompt users for the information. Execute the script. Be sure to use the sequence to add the member numbers.

First Name	Last Name	Address	City	Phone	Join Date
Carmen	Velasquez	283 King Street	Seattle	206-899-6666	08-MAR-90
LaDoris	Ngao	5 Modrany	Bratislava	586-355-8882	08-MAR-90
Midori	Nagayama	68 Via Centrale	Sao Paolo	254-852-5764	17-JUN-91
Mark	Lewis	6921 King Way	Lagos	63-559-7777	07-APR-90
Audry	Ropeburn	86 Chu Street	Hong Kong	41-559-87	18-JAN-91
Molly	Urguhart	3035 Laurier	Quebec	418-542-9988	18-JAN-91

```
SQL> EDIT p15q4b.sql
SET ECHO OFF
SET VERIFY OFF
INSERT INTO member(member_id, first_name, last_name, address,
          city, phone, join_date)
VALUES (member_id_seq.NEXTVAL, '&first_name', '&last_name',
   '&address', '&city', '&phone', '&join_date')
/
COMMIT
/
SET VERIFY ON
SET ECHO ON
SQL> START p15q4b.sql
```

Practice 15 Solutions (continued)

c. Add the following movie copies in the TITLE_COPY table:

Title	Copy Number	Status
Willie and Christmas Too	1	AVAILABLE
Alien	1	AVAILABLE
	2	RENTAL
The Glob	1	AVAILABLE
My Day Off	1	AVAILABLE
	2	AVAILABLE
	3	RENTAL
Miracles on Ice	1	AVAILABLE
Soda Gang	1	AVAILABLE

```
SQL> INSERT INTO title_copy(copy_id, title_id, status)
  2  VALUES (1, 92, 'AVAILABLE');
SQL> INSERT INTO title_copy(copy_id, title_id, status)
  2  VALUES (1, 93, 'AVAILABLE');
SQL> INSERT INTO title_copy(copy_id, title_id, status)
  2  VALUES (2, 93, 'RENTED');
SQL> INSERT INTO title_copy(copy_id, title_id, status)
  2  VALUES (1, 94, 'AVAILABLE');
SQL> INSERT INTO title_copy(copy_id, title_id, status)
  2  VALUES (1, 95, 'AVAILABLE');
SQL> INSERT INTO title_copy(copy_id, title_id,status)
  2  VALUES (2, 95, 'AVAILABLE');
SQL> INSERT INTO title_copy(copy_id, title_id,status)
  2  VALUES (3, 95, 'RENTED');
SQL> INSERT INTO title_copy(copy_id, title_id,status)
  2  VALUES (1, 96, 'AVAILABLE');
SQL> INSERT INTO title_copy(copy_id, title_id,status)
  2  VALUES (1, 97, 'AVAILABLE');
```

d. Add the following rentals to the RENTAL table:
Note: Title number may be different depending on sequence number.

Title	Copy_ number	Customer	Date_ Rented	Date_return_expected	Date_ returned
92	1	101	3 days ago	1 day ago	2 days ago
93	2	101	1 day ago	1 day from now	
95	3	102	2 days ago	Today	
97	1	106	4 days ago	2 days ago	2 days ago

```
SQL> INSERT INTO rental(title_id, copy_id, member_id,
  2              book_date, exp_ret_date, act_ret_date)
  3  VALUES (92, 1, 101, sysdate-3, sysdate-1, sysdate-2);
SQL> INSERT INTO rental(title_id, copy_id, member_id,
  2              book_date, exp_ret_date, act_ret_date)
  3 VALUES  (93, 2, 101, sysdate-1, sysdate-1, NULL);
SQL> INSERT INTO rental(title_id, copy_id, member_id,
  2              book_date, exp_ret_date, act_ret_date)
  3 VALUES  (95, 3, 102, sysdate-2, sysdate, NULL);
SQL> INSERT INTO rental(title_id, copy_id, member_id,
  2              book_date, exp_ret_date,act_ret_date)
  3  VALUES (97, 1, 106, sysdate-4, sysdate-2, sysdate-2);
SQL> COMMIT;
```

Practice 15 Solutions (continued)

5. Create a view named TITLE_AVAIL to show the movie titles and the availability of each copy and its expected return date if rented. Query all rows from the view.

```
SQL> CREATE VIEW title_avail AS
  2    SELECT   t.title, c.copy_id, c.status, r.exp_ret_date
  3    FROM     title t, title_copy c, rental r
  4    WHERE    t.title_id = c.title_id
  5    AND      c.copy_id = r.copy_id(+)
  6    AND      c.title_id = r.title_id(+);

SQL> SELECT     *
  2  FROM       title_avail;
  3  ORDER BY title, copy_id;
```

6. Make changes to data in the tables.

   a. Add a new title. The movie is “Interstellar Wars,” which is rated PG and classified as a Sci-fi movie. The release date is 07-JUL-77. The description is “Futuristic interstellar action movie. Can the rebels save the humans from the evil Empire?” Be sure to add a title copy record for two copies.

```
SQL> INSERT INTO title(title_id, title, description, rating,
  2              category, release_date)
  3  VALUES (title_id_seq.NEXTVAL, 'Interstellar Wars',
  4         'Futuristic interstellar action movie.  Can the
  5              rebels save the humans from the evil Empire?',
  6              'PG', 'SCIFI', '07-JUL-77');
```

   b. Enter two reservations. One reservation is for Carmen Velasquez, who wants to rent “Interstellar Wars.” The other is for Mark Lewis, who wants to rent “Soda Gang.”

```
SQL> INSERT INTO reservation (res_date, member_id, title_id)
  2  VALUES (SYSDATE, 101, 98);
SQL> INSERT INTO reservation (res_date, member_id, title_id)
  2  VALUES (SYSDATE, 101, 99);
```

c. Customer Carmen Velasquez rents the movie “Interstellar Wars,” copy 1. Remove her reservation for the movie. Record the information about the rental. Allow the default value for the expected return date to be used. Verify that the rental was recorded by using the view you created.

```
SQL> INSERT INTO rental(title_id, copy_id, member_id)
  2  VALUES (98, 1,101);
SQL> UPDATE    title_copy
  2  SET       status= 'RENTED'
  3  WHERE     title_id = 98
  4  AND       copy_id = 1;
SQL> DELETE
  2  FROM      reservation
  3  WHERE     member_id = 101;
SQL> SELECT    *
  2  FROM      title_avail
  3  ORDER BY  title, copy_id;
```

7. Make a modification to one of the tables.

a. Add a PRICE column to the TITLE table to record the purchase price of the video. The column should have a total length of eight digits and two decimal places. Verify your modifications.

```
SQL> ALTER TABLE title
  2  ADD (price         NUMBER(8,2));
SQL> DESCRIBE title
```

b. Create a script named *p15q7b.sql* to update each video with a price according to the following list. Note: Have the title id numbers available for this exercise.

Title	Price
Willie and Christmas Too	25
Alien Again	35
The Glob	35
My Day Off	35
Miracles on Ice	98
Soda Gang	35
Interstellar Wars	29

```
SET ECHO OFF
SET VERIFY OFF
UPDATE        title
SET           price = &price
WHERE         title_id = &title_id
/
SET VERIFY OFF
SET ECHO OFF
SQL> START p15q7b.sql
```

c. Ensure that in the future all titles will contain a price value. Verify the constraint.

```
SQL> ALTER TABLE title
  2  MODIFY (price CONSTRAINT title_price_nn NOT NULL);
SQL> SELECT   constraint_name, constraint_type,
  2           search_condition
  3  FROM     user_constraints
  4  WHERE    table_name = 'TITLE';
```

Practice 15 Solutions (continued)

8. Create a report titles Customer History Report. This report will contain each customer's history of renting videos. Be sure to include the customer name, movie rented, dates of the rental, and duration of rentals. Total the number of rentals for all customers for the reporting period. Save the script in a file name *p15q8.sql.*

```
SQL> EDIT p15q8.sql
SET ECHO OFF
SET VERIFY OFF
SET PAGESIZE 30
COLUMN   member  FORMAT A17
COLUMN   title FORMAT A15
COLUMN   book_date FORMAT A9
COLUMN   duration FORMAT 9999999
BREAK ON member  SKIP 1 ON REPORT
SELECT      m.first_name||' '||m.last_name MEMBER, t.title,
            r.book_date, r.act_ret_date - r.book_date DURATION
FROM        member m, title t, rental r
WHERE       r.member_id = m.member_id
AND         r.title_id = t.title_id
ORDER BY    member
/
CLEAR BREAK
COLUMN member CLEAR
COLUMN title CLEAR
COLUMN book_date CLEAR
COLUMN duration CLEAR
SET VERIFY ON
SET PAGESIZE 24
SET ECHO ON
```

Practice 16 Solutions

1. Evaluate each of the following declarations. Determine which of them are *not* legal and explain why.

   a.
```
DECLARE
 v_id              NUMBER(4);
```
Legal.

   b.
```
DECLARE
  v_x, v_y, v_z    VARCHAR2(10);
```
Illegal because only one identifier per declaration is allowed.

   c.
```
DECLARE
  v_birthdate      DATE NOT NULL;
```
Illegal because the NOT NULL variable must be initialized.

   d.
```
DECLARE
  v_in_stock       BOOLEAN := 1;
```
Illegal because 1 is not a Boolean expression.

   e.
```
DECLARE
  TYPE name_table_type IS TABLE OF VARCHAR2(20)
    INDEX BY BINARY_INTEGER;
  dept_name_table name_table_type;
```
Legal.

Practice 16 Solutions (continued)

2. In each of the following assignments, determine the data type of the resulting expression.

a.
```
v_days_to_go    := v_due_date - SYSDATE;
```
Number

b.
```
v_sender   := USER || ': ' || TO_CHAR(v_dept_no);
```
Character string

c.
```
v_sum      := $100,000 + $250,000;
```
Illegal, PL/SQL cannot convert special symbols from VARCHAR2 to NUMBER

d.
```
v_flag     := TRUE;
```
Boolean

e.
```
v_n1       := v_n2 > (2 * v_n3);
```
Boolean

f.
```
v_value    := NULL;
```
Any scalar data type

3. Create an anonymous block to output the phrase "My PL/SQL Block Works" to the screen.

```
VARIABLE g_message VARCHAR2(30)
BEGIN
  :g_message := 'My PL/SQL Block Works';
END;
/
PRINT g_message
SQL> START p16q3.sql
```

```
G_MESSAGE
-----------------------------------------------------
My PL/SQL Block Works
```

Practice 16 Solutions (continued)

If you have time, complete the following exercise.

4. Create a block that declares two variables. Assign the value of these PL/SQL variables to SQL*Plus host variables and print the results of the PL/SQL variables to the screen. Execute your PL/SQL block. Save your PL/SQL block to a file named *p16q4.sql.*

```
V_CHAR Character (variable length)
V_NUM  Number
```

Assign values to these variables as follows:

```
Variable         Value
----------  ----------------------------------------
V_CHAR      The literal '42 is the answer'
V_NUM       The first two characters from V_CHAR
```

```
VARIABLE g_char VARCHAR2(30)
VARIABLE g_num  NUMBER
DECLARE
  v_char VARCHAR2(30);
v_num  NUMBER(11,2);
BEGIN
  v_char := '42 is the answer';
  v_num  := TO_NUMBER(SUBSTR(v_char,1,2));
  :g_char := v_char;
  :g_num  := v_num;
END;
/
PRINT g_char
PRINT g_num
SQL> START p16q4.sql
```

Practice 17 Solutions

PL/SQL Block

```
DECLARE
  v_weight   NUMBER(3) := 600;
  v_message  VARCHAR2(255) := 'Product 10012';
BEGIN

                 SUBBLOCK
    DECLARE
      v_weight    NUMBER(3) := 1;
      v_message   VARCHAR2(255) := 'Product 11001';
      v_new_locn  VARCHAR2(50) := 'Europe';
    BEGIN
      v_weights := v_weight + 1;
      v_new_locn := 'Western ' || v_new_locn;
    END;

  v_weight := v_weight + 1;
  v_message := v_message || ' is in stock';
  v_new_locn := 'Western ' || v_new_locn;

END;
```

Practice 17 Solutions (continued)

1. Evaluate the PL/SQL block on the previous page and determine each of the following values according to the rules of scoping.

   a. The value of V_WEIGHT in the subblock is

      "2" and the data type is NUMBER.

   b. The value of V_NEW_LOCN in the subblock is

      "Western Europe" and the data type is VARCHAR2.

   c. The value of V_WEIGHT in the main block is

      "601" and the data type is NUMBER.

   d. The value of V_MESSAGE in the main block is

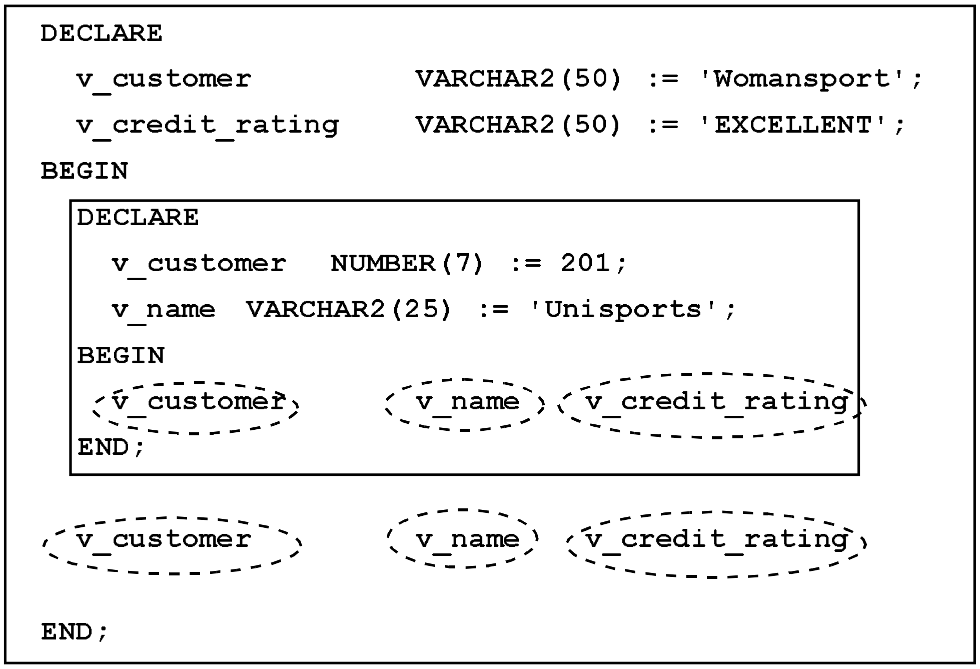
      "Product 10012 is in stock" and the data type is VARCHAR2.

   e. The value of V_NEW_LOCN in the main block is

      Illegal because v_new_locn is not visible outside the subblock.

Practice 17 Solutions (continued)

Scope Example



```
DECLARE
  v_customer          VARCHAR2(50)  := 'Womansport';
  v_credit_rating     VARCHAR2(50)  := 'EXCELLENT';
BEGIN
  DECLARE
    v_customer   NUMBER(7)  := 201;
    v_name  VARCHAR2(25)  := 'Unisports';
  BEGIN
    v_customer        v_name   v_credit_rating
  END;

  v_customer          v_name    v_credit_rating

END;
```

Practice 17 Solutions (continued)

2. Suppose you embed a subblock within a block, as shown on the previous page. You declare two variables, V_CUSTOMER and V_CREDIT_RATING, in the main block. You also declare two variables, V_CUSTOMER and V_NAME, in the subblock. Determine the values for each of the following cases.

   a. The value of V_CUSTOMER in the subblock is

      “201” and the data type is NUMBER.

   b. The value of V_NAME in the subblock is

      “Unisports” and the data type is VARCHAR2.

   c. The value of V_CREDIT_RATING in the subblock is

      “EXCELLENT” and the data type is VARCHAR2.

   d. The value of V_CUSTOMER in the main block is

      “Womansport” and the data type is VARCHAR2.

   e. The value of V_NAME in the main block is

      V_NAME is not visible in the main block and you would see an error.

   f. The value of V_CREDIT_RATING in the main block is

      “EXCELLENT” and the data type is VARCHAR2.

Practice 17 Solutions (continued)

3. Create and execute a PL/SQL block that accepts two numbers through SQL*Plus substitution variables. The first number should be divided by the second number and have the second number added to the result. The result should be written to a PL/SQL variable and printed to the screen.

```
SET VERIFY OFF
VARIABLE v_result NUMBER
ACCEPT p_num1 PROMPT 'Please enter the first number: '
ACCEPT p_num2 PROMPT 'Please enter the second number: '
DECLARE
  v_num1    NUMBER(9,2) := &p_num1;
  v_num2    NUMBER(9,2) := &p_num2;
BEGIN
 :v_result := (v_num1/v_num2) + v_num2;
END;
/
PRINT v_result
SET VERIFY ON
SQL> START p17q3.sql
```

```
ACCEPT p_num1 PROMPT 'Please enter the first number: '
ACCEPT p_num2 PROMPT 'Please enter the second number: '
DECLARE
  v_num1    NUMBER(9,2) := &p_num1;
  v_num2    NUMBER(9,2) := &p_num2;
BEGIN
 dbms_output.put_line(TO_CHAR(v_num1/v_num2) + v_num2);
END;
/
```

Practice 17 Solutions (continued)

4. Build a PL/SQL block that computes the total compensation for one year. The annual salary and the annual bonus percentage are passed to the PL/SQL block through SQL*Plus substitution variables and the bonus needs to be converted from a whole number to a decimal (for example, 15 to .15). If the salary is null, set it to zero before computing the total compensation. Execute the PL/SQL block. Reminder: Use the NVL function to handle null values.

   Note: To test the NVL function type NULL at the prompt; pressing [Return] results in a missing expression error.

```
SET VERIFY OFF
VARIABLE g_total NUMBER
ACCEPT p_salary PROMPT 'Please enter the salary amount: '
ACCEPT p_bonus PROMPT 'Please enter the bonus percentage: '
DECLARE
  v_salary  NUMBER := &p_salary;
  v_bonus   NUMBER := &p_bonus;
BEGIN
  :g_total := NVL(v_salary, 0) * (1 + NVL(v_bonus, 0) / 100);
END;
/
PRINT g_total
SET VERIFY ON
SQL> START p17q4.sql
```

```
ACCEPT p_salary PROMPT 'Please enter the salary amount: '
ACCEPT p_bonus PROMPT 'Please enter the bonus percentage: '
DECLARE
  v_salary  NUMBER := &p_salary;
  v_bonus   NUMBER := &p_bonus;
BEGIN
  dbms_output.put_line(TO_CHAR(NVL(v_salary, 0) *
                      (1 + NVL(v_bonus, 0) / 100)));
END;
/
```

Practice 18 Solutions

1. Create a PL/SQL block that selects the maximum department number in the DEPT table and store it in a SQL*Plus variable. Print the results to the screen. Save your PL/SQL block to a file named *p18q1.sql.*

```
VARIABLE g_max_deptno NUMBER
DECLARE
  v_max_deptno   NUMBER;
BEGIN
  SELECT     MAX(deptno)
  INTO       v_max_deptno
  FROM       dept;
  :g_max_deptno := v_max_deptno;
END;
/
PRINT g_max_deptno
SQL> START p18q1.sql
```

```
DECLARE
  v_max_deptno   NUMBER;
BEGIN
  SELECT     MAX(deptno)
  INTO       v_max_deptno
  FROM       dept;
  dbms_output.put_line(TO_CHAR(v_max_deptno));
END;
/
```

2. Create a PL/SQL block that inserts a new department into the DEPT table. Save your PL/SQL block to a file named *p18q2.sql.*

   a. Use the department number retrieved from exercise 1 and add 10 to that number as the input department number for the new department.

   b. Use a parameter for the department name.

   c. Leave the location null for now.

Practice 18 Solutions (continued)

```
SET VERIFY OFF
ACCEPT p_deptno PROMPT 'Please enter the department number: '
ACCEPT p_dept_name PROMPT 'Please enter the department name: '
BEGIN
  INSERT INTO dept (deptno, dname, loc)
  VALUES (&p_deptno, '&p_dept_name', NULL);
  COMMIT;
END;
/
SET VERIFY ON
```

d. Execute the PL/SQL block.

```
SQL> START p18q2.sql
```

e. Display the new department that you created.

```
SELECT   *
FROM     dept
WHERE    deptno = :g_max_deptno + 10;
```

3. Create a PL/SQL block that updates the location for an existing department. Save your PL/SQL block to a file named *p18q3.sql.*

a. Use a parameter for the department number.

b. Use a parameter for the department location.

c. Test the PL/SQL block.

d. Display the department number, department name, and location for the updated department.

```
SET VERIFY OFF
ACCEPT p_deptno PROMPT 'Please enter the department number: '
ACCEPT p_loc PROMPT 'Please enter the department location: '
BEGIN
  UPDATE dept
  SET    loc = '&p_loc'
  WHERE  deptno = &p_deptno;
COMMIT;
END;
/
SET VERIFY ON
SQL> START p18q3.sql
```

Practice 18 Solutions (continued)

e. Display the department that you updated.

```
SQL> SELECT *
  2  FROM   dept
  3  WHERE  deptno = &p_deptno;
```

4. Create a PL/SQL block that deletes the department created in exercise 2. Save your PL/SQL block to a file named *p18q4.sql.*

a. Use a parameter for the department number.

b. Print to the screen the number of rows affected.

c. Test the PL/SQL block.

```
SET VERIFY OFF
VARIABLE g_result VARCHAR2(40)
ACCEPT p_deptno PROMPT 'Please enter the department number: '
DECLARE
  v_result  NUMBER(2);
BEGIN
  DELETE
  FROM      dept
  WHERE          deptno = &p_deptno;
  v_result := SQL%ROWCOUNT;
  :g_result := (TO_CHAR(v_result) || ' row(s) deleted.');
  COMMIT;
END;
/
SET VERIFY ON
PRINT g_result
SQL> START p18q4.sql
```

```
ACCEPT p_deptno PROMPT 'Please enter the department number: '
DECLARE
  v_result  NUMBER(2);
BEGIN
  DELETE
  FROM      dept
  WHERE     deptno = &p_deptno;
  v_result := SQL%ROWCOUNT;
  dbms_output.put_line(TO_CHAR(v_result)||
 ' row(s) deleted.');
  COMMIT;
END;
/
```

Practice 18 Solutions (continued)

d. What happens if you enter a department number that does not exist?

If the operator enters a department number that does not exist, the PL/SQL block completes successfully because this does not constitute an exception.

e. Confirm that the department has been deleted.

```
SQL> SELECT   *
  2   FROM   dept
  3   WHERE  deptno = &p_deptno;
```

Practice 19 Solutions

1. Run the script LABS*lab19_1.sql* to create the MESSAGES table. Write a PL/SQL block to insert numbers into the MESSAGES table.

```
CREATE TABLE messages (results VARCHAR2 (60))
/
```

a. Insert the numbers 1 to 10 excluding 6 and 8.

b. Commit before the end of the block.

```
BEGIN
FOR i IN 1..10 LOOP
  IF i = 6 or i = 8 THEN
    null;
  ELSE
 INSERT INTO messages(results)
 VALUES (i);
  END IF;
  COMMIT;
END LOOP;
END;
/
```

c. Select from the MESSAGES table to verify that your PL/SQL block worked.

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

2. Create a PL/SQL block that computes the commission amount for a given employee based on the employee's salary.

a. Run the script LABS*lab19_2.sql* to insert a new employee into the EMP table.
Note: The employee will have a NULL salary.

```
SQL> START lab19_2.sql
```

b. Accept the employee number as user input with a SQL*Plus substitution variable.

c. If the employee's salary is less than $1,000, set the commission amount for the employee to 10% of the salary.

d. If the employee's salary is between $1,000 and $1,500, set the commission amount for the employee to 15% of the salary.

e. If the employee's salary exceeds $1,500, set the commission amount for the employee to 20% of the salary.

f. If the employee's salary is NULL, set the commission amount for the employee to 0.

g. Commit.

Practice 19 Solutions (continued)

```
ACCEPT p_empno PROMPT 'Please enter employee number: '
DECLARE
  v_empno         emp.empno%TYPE := &p_empno;
  v_sal           emp.sal%TYPE;
  v_comm          emp.comm%TYPE;
BEGIN
  SELECT sal
  INTO v_sal
  FROM emp
  WHERE empno = v_empno;
  IF v_sal < 1000 THEN
    v_comm := .10;
   ELSIF v_sal BETWEEN 1000 and 1500 THEN
    v_comm := .15;
  ELSIF v_sal > 1500 THEN
    v_comm := .20;
  ELSE
    v_comm := 0;
END IF;
  UPDATE emp
  SET comm = sal * v_comm
  WHERE empno = v_empno;
COMMIT;
END;
/
```

h. Test the PL/SQL block for each case using the following test cases, and check each updated commission.

Employee Number	Salary	Resulting Commission
7369	800	80
7934	1300	195
7499	1600	320
8000	NULL	NULL

```
SQL> SELECT       empno, sal, comm
  2  FROM         emp
  3  WHERE        empno IN (7369, 7934,7499, 8000)
  4  ORDER BY     comm;
```

If you have time, complete the following exercises.

3. Modify *p16q4.sql* to insert the text "Number is odd" or "Number is even," depending on whether the value is odd or even, into the MESSAGES table. Query the MESSAGES table to determine if your PL/SQL block worked.

```
DECLARE
  v_char VARCHAR2(30);
  v_num  NUMBER(11,2);
BEGIN
  v_char := '42 is the answer';
  v_num  := TO_NUMBER(SUBSTR(v_char,1,2));
  IF mod(v_num, 2) = 0 THEN
    INSERT INTO messages (results)
    VALUES ('Number is even');
  ELSE
    INSERT INTO messages (results)
    VALUES ('Number is odd');
  END IF;
END;
/
SQL> SELECT *
  2  FROM messages;
```

4. Add a new column to the EMP table for storing asterisk (*).

```
SQL> ALTER TABLE  emp
  2  ADD stars    VARCHAR2(100);
```

5. Create a PL/SQL block that rewards an employee by appending an asterisk in the STARS column for every $100 of the employee's salary. Round the employee's salary to the nearest whole number. Save your PL/SQL block to a file called *p19q5.sql.*
   a. Accept the employee ID as user input with a SQL*Plus substitution variable.
   b. Create a variable to hold a string of asterisks and initialize it to NULL.
   c. Append an asterisk to the string for every $100 of the salary amount. For example, if the employee has a salary amount of $800, the string of asterisks should contain eight asterisks.
   d. Update the STARS column for the employee with the string of asterisks.

e. Commit.

f. Test the block for employees who have no salary and for an employee who has a salary.

```
SET VERIFY OFF
ACCEPT p_empno PROMPT 'Please enter the employee number: '
DECLARE
  v_empno    emp.empno%TYPE := &p_empno;
  v_asterisk emp.stars%TYPE := NULL;
  v_sal      emp.sal%TYPE;
BEGIN
  SELECT NVL(ROUND(sal/100), 0)
  INTO v_sal
  FROM emp
  WHERE empno = v_empno;
  FOR i IN 1..v_sal LOOP
    v_asterisk := v_asterisk ||'*';
  END LOOP;
  UPDATE emp
  SET stars = v_asterisk
  WHERE empno = v_empno;
  COMMIT;
END;
/
SET VERIFY ON
SQL> START p19q5.sql
SQL> SELECT  empno, sal, stars
  2  FROM    emp
  3  WHERE   empno IN (7934, 8000);
```

Practice 20 Solutions

1. Run the script LABS\lab20_1.sql to create a new table for storing employees and their salaries.

```
SQL> START lab20_1.sql
```

2. Write a PL/SQL block to retrieve the name and salary of a given employee from the EMP table based on the employee's number, incorporate PL/SQL tables.
   a. Declare two PL/SQL tables, ENAME_TABLE and SAL_TABLE, to temporarily store the names and salaries.
   b. As each name and salary is retrieved within the loop, store them in the PL/SQL tables.
   c. Outside the loop, transfer the names and salaries from the PL/SQL tables into the TOP_DOGS table.
   d. Empty the TOP_DOGS table and test the practice.

```
SQL> DELETE
  2  FROM top_dogs;
```

```
SET VERIFY OFF
ACCEPT p_empno PROMPT 'Please enter the employee number: '
DECLARE
  TYPE ename_table_type   IS TABLE OF VARCHAR2(10)
    INDEX BY BINARY_INTEGER;
  TYPE sal_table_type IS TABLE OF NUMBER(7,2)
    INDEX BY BINARY_INTEGER;
    v_empno      emp.empno%TYPE := &p_empno;
    v_ename      emp.ename%TYPE;
    v_sal        emp.sal%TYPE;
    ename_table  ename_table_type;
    sal_table    sal_table_type;
    i            BINARY_INTEGER := 0;
BEGIN
  DELETE
  FROM top_dogs;
  SELECT    ename, sal
  INTO      v_ename, v_sal
  FROM      emp
  WHERE     empno = &p_empno;
  ename_table(i) := v_ename;
  sal_table(i)   := v_sal;
  INSERT INTO top_dogs (name, salary)
  VALUES (ename_table(i), sal_table(i));
  COMMIT;
END;
/
SET VERIFY ON

SQL> START p20q2.sql
SQL> SELECT *
  2  FROM   top_dogs;
```

Practice 21 Solutions

1. Create a PL/SQL block that determines the top employees with respect to salaries.
   a. Accept a number *n* as user input with a SQL*Plus substitution parameter.
   b. In a loop, get the last names and salaries of the top *n* people with respect to salary in the EMP table.
   c. Store the names and salaries in the TOP_DOGS table.
   d. Assume that no two employees have the same salary.
   e. Test a variety of special cases, such a *n* = 0, where *n* is greater than the number of employees in the EMP table. Empty the TOP_DOGS table after each test.

```
ACCEPT p_num -
PROMPT 'Please enter the number of top money makers: '
DECLARE
  v_num          NUMBER(3) := &p_num;
  v_ename        emp.ename%TYPE;
  v_sal          emp.sal%TYPE;
  CURSOR         emp_cursor IS
    SELECT       ename, sal
    FROM         emp
    WHERE        sal IS NOT NULL
    ORDER BY     sal DESC;
BEGIN
OPEN emp_cursor;
FETCH emp_cursor INTO v_ename, v_sal;
WHILE emp_cursor%ROWCOUNT <= v_num AND
      emp_cursor%FOUND LOOP
  INSERT INTO top_dogs (name, salary)
  VALUES (v_ename, v_sal);
   FETCH emp_cursor INTO v_ename, v_sal;
END LOOP;
CLOSE emp_cursor;
COMMIT;
END;
/
SQL> DELETE
  2  FROM    top_dogs;
SQL> START p21q1.sql
SQL> SELECT  *
  2  FROM    top_dogs;
```

Practice 21 Solutions (continued)

2. Consider the case where several employees have the same salary. If one person is listed, then all people who have the same salary should also be listed.
   a. For example, if the user enters a value of 2 for *n*, then King, Ford and Scott should be displayed. (These employees are tied for second highest salary.)
   b. If the user enters a value of 3, then King, Ford, Scott, and Jones should be displayed.
   c. Delete all rows from TOP_DOGS and test the practice.

```
ACCEPT p_num -
PROMPT 'Please enter the number of top money makers: '
DECLARE
  v_num           NUMBER(3) := &p_num;
  v_ename    emp.ename%TYPE;
  v_current_sal   emp.sal%TYPE;
  v_last_sal emp.sal%TYPE := -1;  --sal never neg
CURSOR        emp_cursor IS
  SELECT      ename, sal
  FROM        emp
  WHERE       sal IS NOT NULL
  ORDER BY    sal DESC;
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor INTO v_ename, v_current_sal;
  WHILE (emp_cursor%ROWCOUNT <= v_num OR
         v_current_sal = v_last_sal) AND
          emp_cursor%FOUND LOOP
    INSERT INTO top_dogs (name, salary)
    VALUES (v_ename, v_current_sal);
    v_last_sal := v_current_sal;
     FETCH emp_cursor INTO v_ename, v_current_sal;
  END LOOP;
  CLOSE emp_cursor;
COMMIT;
END;
/
SQL> DELETE
  2  FROM    top_dogs;
SQL> START p21q2.sql
SQL> SELECT  *
  2  FROM    top_dogs;
```

Practice 22 Solutions

1. Write a query to retrieve all the departments and the employees in each department. Insert the results in the MESSAGES table. Use a cursor to retrieve the department number and pass the department number to a cursor to retrieve the employees in that department.

```
DECLARE
  v_current_deptno     dept.deptno%TYPE;
  v_emp                VARCHAR2(50);
  CURSOR dept_cursor IS
    SELECT   deptno
    FROM     dept
    ORDER BY deptno;
  CURSOR emp_cursor(v_deptno    NUMBER) IS
    SELECT   ename ||' - Department '||TO_CHAR(deptno)
    FROM     emp
    WHERE    deptno = v_deptno;
BEGIN
  OPEN dept_cursor;
  LOOP
    FETCH dept_cursor INTO v_current_deptno;
    EXIT WHEN dept_cursor%NOTFOUND;
    IF emp_cursor%ISOPEN THEN
      CLOSE emp_cursor;
    END IF;
    OPEN emp_cursor (v_current_deptno);
    LOOP
      FETCH emp_cursor INTO v_emp;
      EXIT WHEN emp_cursor%NOTFOUND;
      INSERT INTO messages (results)
      VALUES (v_emp);
    END LOOP;
    CLOSE emp_cursor;
  END LOOP;
  CLOSE dept_cursor;
  COMMIT;
END;
/
SQL> START p22q1.sql
SQL> SELECT  *
  2  FROM    messages;
```

Practice 22 Solutions (continued)

2. Modify *p19q5.sql* to incorporate the FOR UPDATE and WHERE CURRENT OF functionality in cursor processing.

```
SET VERIFY OFF
ACCEPT p_empno PROMPT 'Please enter the employee number: '
DECLARE
  v_empno    emp.empno%TYPE := &p_empno;
  v_asterisk emp.stars%TYPE := NULL;
  CURSOR emp_cursor IS
    SELECT   empno, NVL(ROUND(sal/100), 0) sal
    FROM     emp
    WHERE    empno = v_empno
    FOR UPDATE;
BEGIN
  FOR emp_record IN emp_cursor LOOP
  BEGIN
    FOR i IN 1..emp_record.sal LOOP
      v_asterisk := v_asterisk ||'*';
    END LOOP;
  UPDATE emp
  SET stars = v_asterisk
  WHERE CURRENT OF emp_cursor;
  v_asterisk := NULL;
  END;
  END LOOP;
COMMIT;
END;
/
SET VERIFY ON
SQL> START p22q2.sql
SQL> SELECT  empno, sal, stars
  2  FROM    emp
  3  WHERE   empno IN (7844, 7900, 8000);
```

Practice 23 Solutions

1. Write a PL/SQL block to select the name of the employee with a given salary value.

   a. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the MESSAGES table, the message "More than one employee with a salary of <*salary*>."

   b. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the MESSAGES table, the message "No employee with a salary of <*salary*>."

   c. If the salary entered returns only one row, insert into the MESSAGES table the employee's name and the salary amount.

   d. Handle any other exception with an appropriate exception handler and insert into the MESSAGES table, the message "Some other error occurred."

   e. Test the block for a variety of test cases.

```
SET VERIFY OFF
ACCEPT p_sal PROMPT 'Please enter the salary value: '
DECLARE
  v_ename   emp.ename%TYPE;
  v_sal     emp.sal%TYPE := &p_sal;
BEGIN
  SELECT    ename
  INTO      v_ename
  FROM      emp
  WHERE          sal = v_sal;
  INSERT INTO messages (results)
  VALUES (v_ename || ' - ' || v_sal);
EXCEPTION
  WHEN no_data_found THEN
    INSERT INTO messages (results)
    VALUES ('No employee with a salary of '|| TO_CHAR(v_sal));
  WHEN too_many_rows THEN
    INSERT INTO messages (results)
    VALUES ('More than one employee with a salary of '||
             TO_CHAR(v_sal));
  WHEN others THEN
    INSERT INTO messages (results)
    VALUES ('Some other error occurred.');
END;
/
SET VERIFY ON
SQL> START p23q1.sql
SQL> START p23q1.sql
SQL> START p23q1.sql
```

Practice 23 Solutions (continued)

2. Modify *p18q3.sql* to add an exception handler.

   a. Write an exception handler for the error to pass a message to the user that the specified department does not exist.

   b. Execute the PL/SQL block by entering a department that does not exist.

```
SET VERIFY OFF
VARIABLE g_message VARCHAR2(40)
ACCEPT p_deptno PROMPT 'Please enter the department number: '
ACCEPT p_loc PROMPT 'Please enter the department location: '
DECLARE
  e_invalid_dept EXCEPTION;
  v_deptno       dept.deptno%TYPE := &p_deptno;
BEGIN
  UPDATE dept
  SET loc = '&p_loc'
  WHERE deptno = v_deptno;
  IF SQL%NOTFOUND THEN
    raise e_invalid_dept;
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_dept THEN
    :g_message := 'Department '|| TO_CHAR(v_deptno) ||
                  ' is an invalid department';
END;
/
SET VERIFY ON
PRINT g_message
SQL> START p23q2.sql
```

```
SET VERIFY OFF
ACCEPT p_deptno PROMPT 'Please enter the department number: '
ACCEPT p_loc PROMPT 'Please enter the department location: '
DECLARE
  e_invalid_dept  EXCEPTION;
  v_deptno        dept.deptno%TYPE := &p_deptno;
BEGIN
  UPDATE dept
  SET loc = '&p_loc'
  WHERE deptno = v_deptno;
  IF SQL%NOTFOUND THEN
    raise e_invalid_dept;
  END IF;
  COMMIT;

EXCEPTION
  WHEN e_invalid_dept THEN
    dbms_output.put_line('Department '|| TO_CHAR(v_deptno) ||
                 ' is an invalid department');
END;
/
SET VERIFY ON
```

Practice 23 Solutions (continued)

3. Write a PL/SQL block that prints the names of the employees who make plus or minus $100 of the salary value entered.

   a. If there is no employee within that salary range, print a message to the user indicating that is the case. Use an exception for this case.

   b. If there are one or more employees within that range, the message should indicate how many employees are in that salary range.

   c. Handle any other exception with an appropriate exception handler, the message should indicate that some other error occurred.

```
VARIABLE g_message VARCHAR2(100)
SET VERIFY OFF
ACCEPT p_sal PROMPT 'Please enter the salary: '
DECLARE
  v_sal                 emp.sal%TYPE := &p_sal;
  v_low_sal             emp.sal%TYPE := v_sal - 100;
  v_high_sal            emp.sal%TYPE := v_sal + 100;
  v_no_emp        NUMBER(7);
  e_no_emp_returned     EXCEPTION;
  e_more_than_one_emp EXCEPTION;
BEGIN
  SELECT     count(ename)
  INTO       v_no_emp
  FROM       emp
  WHERE           sal between (v_sal - 100) and (v_sal + 100);
  IF v_no_emp  = 0 THEN
    RAISE e_no_emp_returned;
  ELSIF v_no_emp > 0 THEN
    RAISE e_more_than_one_emp;
  END IF;
EXCEPTION
  WHEN e_no_emp_returned THEN
    :g_message := 'There are no employee salary between '||
                  TO_CHAR(v_low_sal) || ' and '||
                  TO_CHAR(v_high_sal);
  WHEN e_more_than_one_emp THEN
    :g_message := 'There are '|| TO_CHAR(v_no_emp) ||
                  ' employee(s) with a salary between '||
                 TO_CHAR(v_low_sal) || ' and '||
                 TO_CHAR(v_high_sal);
END;
/
SET VERIFY ON
PRINT g_message
SQL> START p23q3.sql
```

Practice 23 Solutions (continued)

```
SET VERIFY OFF
ACCEPT p_sal PROMPT 'Please enter the salary: '
DECLARE
  v_sal           emp.sal%TYPE := &p_sal;
  v_low_sal       emp.sal%TYPE := v_sal - 100;
  v_high_sal      emp.sal%TYPE := v_sal + 100;
  v_no_emp        NUMBER(7);
  e_no_emp_returned    EXCEPTION;
  e_more_than_one_emp  EXCEPTION;
BEGIN
  SELECT count(ename)
  INTO       v_no_emp
  FROM       emp
  WHERE      sal between (v_sal - 100) and (v_sal + 100);
  IF v_no_emp  = 0 THEN
    RAISE e_no_emp_returned;
  ELSIF v_no_emp > 0 THEN
    RAISE e_more_than_one_emp;
  END IF;
EXCEPTION
  WHEN e_no_emp_returned THEN
    dbms_output.put_line('There are no employee salary
                  between '|| TO_CHAR(v_low_sal) || ' and '||
                  TO_CHAR(v_high_sal));
  WHEN e_more_than_one_emp THEN
    dbms_output.put_line('There are '|| TO_CHAR(v_no_emp) ||
                  ' employee(s) with a salary between '||
                  TO_CHAR(v_low_sal) || ' and '||
                  TO_CHAR(v_high_sal));
  WHEN others THEN
    dbms_output.put_line('Some other error occurred.');
END;
/
SET VERIFY ON
```

B

Table Descriptions and Data

EMP Table

```
SQL> DESCRIBE emp
```

```
Name                            Null?    Type
------------------------------- -------- ----
EMPNO                           NOT NULL NUMBER(4)
ENAME                                    VARCHAR2(10)
JOB                                      VARCHAR2(9)
MGR                                      NUMBER(4)
HIREDATE                                 DATE
SAL                                      NUMBER(7,2)
COMM                                     NUMBER(7,2)
DEPTNO                          NOT NULL NUMBER(2)
```

```
SQL> SELECT * FROM emp;
```

```
    EMPNO ENAME      JOB             MGR HIREDATE        SAL      COMM    DEPTNO
--------- ---------- --------- --------- --------- --------- --------- ---------
     7839 KING       PRESIDENT           17-NOV-81      5000                  10
     7698 BLAKE      MANAGER        7839 01-MAY-81      2850                  30
     7782 CLARK      MANAGER        7839 09-JUN-81      2450                  10
     7566 JONES      MANAGER        7839 02-APR-81      2975                  20
     7654 MARTIN     SALESMAN       7698 28-SEP-81      1250      1400        30
     7499 ALLEN      SALESMAN       7698 20-FEB-81      1600       300        30
     7844 TURNER     SALESMAN       7698 08-SEP-81      1500         0        30
     7900 JAMES      CLERK          7698 03-DEC-81       950                  30
     7521 WARD       SALESMAN       7698 22-FEB-81      1250       500        30
     7902 FORD       ANALYST        7566 03-DEC-81      3000                  20
     7369 SMITH      CLERK          7902 17-DEC-80       800                  20
     7788 SCOTT      ANALYST        7566 09-DEC-82      3000                  20
     7876 ADAMS      CLERK          7788 12-JAN-83      1100                  20
     7934 MILLER     CLERK          7782 23-JAN-82      1300                  10
```

DEPT Table

```
SQL> DESCRIBE dept
```

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

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

```
   DEPTNO DNAME          LOC

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

SALGRADE Table

```
SQL> DESCRIBE salgrade
```

```
Name                            Null?    Type
------------------------------- -------- ----
GRADE                                    NUMBER
LOSAL                                    NUMBER
HISAL                                    NUMBER
```

```
SQL> SELECT * FROM salgrade;
```

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

ORD Table

```
SQL> DESCRIBE ord
```

```
Name                            Null?    Type
------------------------------- -------- ----
ORDID                           NOT NULL NUMBER(4)
ORDERDATE                                DATE
COMMPLAN                                 VARCHAR2(1)
CUSTID                          NOT NULL NUMBER(6)
SHIPDATE                                 DATE
TOTAL                                    NUMBER(8,2)
```

```
SQL> SELECT * FROM ord;
```

```
    ORDID ORDERDATE C    CUSTID SHIPDATE      TOTAL
--------- --------- - --------- --------- ---------
      610 07-JAN-87 A       101 08-JAN-87     101.4
      611 11-JAN-87 B       102 11-JAN-87        45
      612 15-JAN-87 C       104 20-JAN-87      5860
      601 01-MAY-86 A       106 30-MAY-86       2.4
      602 05-JUN-86 B       102 20-JUN-86        56
      604 15-JUN-86 A       106 30-JUN-86       698
      605 14-JUL-86 A       106 30-JUL-86      8324
      606 14-JUL-86 A       100 30-JUL-86       3.4
      609 01-AUG-86 B       100 15-AUG-86      97.5
      607 18-JUL-86 C       104 18-JUL-86       5.6
      608 25-JUL-86 C       104 25-JUL-86      35.2
      603 05-JUN-86         102 05-JUN-86       224
      620 12-MAR-87         100 12-MAR-87      4450
      613 01-FEB-87         108 01-FEB-87      6400
      614 01-FEB-87         102 05-FEB-87     23940
      616 03-FEB-87         103 10-FEB-87       764
      619 22-FEB-87         104 04-FEB-87      1260
      617 05-FEB-87         105 03-MAR-87     46370
      615 01-FEB-87         107 06-FEB-87       710
      618 15-FEB-87 A       102 06-MAR-87    3510.5
      621 15-MAR-87 A       100 01-JAN-87       730
```

PRODUCT Table

```
SQL> DESCRIBE product
```

```
Name                              Null?    Type
------------------------------- -------- ----
 PRODID                           NOT NULL NUMBER(6)
 DESCRIP                                   VARCHAR2(30)
```

```
SQL> SELECT * FROM product;
```

```
   PRODID DESCRIP
--------- ------------------------------
   100860 ACE TENNIS RACKET I
   100861 ACE TENNIS RACKET II
   100870 ACE TENNIS BALLS-3 PACK
   100871 ACE TENNIS BALLS-6 PACK
   100890 ACE TENNIS NET
   101860 SP TENNIS RACKET
   101863 SP JUNIOR RACKET
   102130 RH: "GUIDE TO TENNIS"
   200376 SB ENERGY BAR-6 PACK
   200380 SB VITA SNACK-6 PACK
```

ITEM Table

```
SQL> DESCRIBE item
```

```
 Name                            Null?    Type
------------------------------- -------- -----------
ORDID                           NOT NULL NUMBER(4)
ITEMID                          NOT NULL NUMBER(4)
PRODID                                   NUMBER(6)
ACTUALPRICE                              NUMBER(8,2)
QTY                                      NUMBER(8)
ITEMTOT                                  NUMBER(8,2)
```

```
SQL> SELECT * FROM item;
```

```
    ORDID    ITEMID    PRODID ACTUALPRICE       QTY   ITEMTOT
--------- --------- --------- ----------- --------- ---------
      610         3    100890          58         1        58
      611         1    100861          45         1        45
      612         1    100860          30       100      3000
      601         1    200376         2.4         1       2.4
      602         1    100870         2.8        20        56
      604         1    100890          58         3       174
      604         2    100861          42         2        84
      604         3    100860          44        10       440
      603         2    100860          56         4       224
      610         1    100860          35         1        35
      610         2    100870         2.8         3       8.4
      613         4    200376         2.2       200       440
      614         1    100860          35       444     15540
      614         2    100870         2.8      1000      2800
      612         2    100861        40.5        20       810
      612         3    101863          10       150      1500
      620         1    100860          35        10       350
      620         2    200376         2.4      1000      2400
      620         3    102130         3.4       500      1700
      613         1    100871         5.6       100       560
      613         2    101860          24       200      4800
      613         3    200380           4       150       600
      619         3    102130         3.4       100       340
      617         1    100860          35        50      1750
      617         2    100861          45       100      4500
      614         3    100871         5.6      1000      5600
```

Continued on next page

ITEM Table (continued)

```
 ORDID     ITEMID     PRODID  ACTUALPRICE        QTY    ITEMTOT
------ ---------- ---------- ------------ ---------- ----------
   616          1     100861           45         10        450
   616          2     100870          2.8         50        140
   616          3     100890           58          2        116
   616          4     102130          3.4         10         34
   616          5     200376          2.4         10         24
   619          1     200380            4        100        400
   619          2     200376          2.4        100        240
   615          1     100861           45          4        180
   607          1     100871          5.6          1        5.6
   615          2     100870          2.8        100        280
   617          3     100870          2.8        500       1400
   617          4     100871          5.6        500       2800
   617          5     100890           58        500      29000
   617          6     101860           24        100       2400
   617          7     101863         12.5        200       2500
   617          8     102130          3.4        100        340
   617          9     200376          2.4        200        480
   617         10     200380            4        300       1200
   609          2     100870          2.5          5       12.5
   609          3     100890           50          1         50
   618          1     100860           35         23        805
   618          2     100861        45.11         50     2255.5
   618          3     100870           45         10        450
   621          1     100861           45         10        450
   621          2     100870          2.8        100        280
   615          3     100871            5         50        250
   608          1     101860           24          1         24
   608          2     100871          5.6          2       11.2
   609          1     100861           35          1         35
   606          1     102130          3.4          1        3.4
   605          1     100861           45        100       4500
   605          2     100870          2.8        500       1400
   605          3     100890           58          5        290
   605          4     101860           24         50       1200
   605          5     101863            9        100        900
   605          6     102130          3.4         10         34
   612          4     100871          5.5        100        550
   619          4     100871          5.6         50        280
```

CUSTOMER Table

```
SQL> DESCRIBE customer
```

```
Name                            Null?    Type
------------------------------- -------- ----
CUSTID                          NOT NULL NUMBER(6)
NAME                                     VARCHAR2(45)
ADDRESS                                  VARCHAR2(40)
CITY                                     VARCHAR2(30)
STATE                                    VARCHAR2(2)
ZIP                                      VARCHAR2(9)
AREA                                     NUMBER(3)
PHONE                                    VARCHAR2(9)
REPID                           NOT NULL NUMBER(4)
CREDITLIMIT                              NUMBER(9,2)
COMMENTS                                 LONG
```

CUSTOMER Table (continued)

```
SQL> SELECT * FROM customer;
```

```
CUSTID  NAME                                          ADDRESS
------- --------------------------------------------- ------------------
    100 JOCKSPORTS                                    345 VIEWRIDGE
    101 TKB SPORT SHOP                                490 BOLI RD.
    102 VOLLYRITE                                     9722 HAMILTON
    103 JUST TENNIS                                   HILLVIEW MALL
    104 EVERY MOUNTAIN                                574 SURRY RD.
    105 K + T SPORTS                                  3476 EL PASEO
    106 SHAPE UP                                      908 SEQUOIA
    107 WOMENS SPORTS                                 VALCO VILLAGE
    108 NORTH WOODS HEALTH AND FITNESS SUPPLY CENTER 98 LONE PINE WAY


 CITY            ST ZIP            AREA PHONE         REPID CREDITLIMIT
 --------------- -- --------- --------- --------- --------- -----------
 BELMONT         CA 96711           415 598-6609       7844        5000
 REDWOOD CITY    CA 94061           415 368-1223       7521       10000
 BURLINGAME      CA 95133           415 644-3341       7654        7000
 BURLINGAME      CA 97544           415 677-9312       7521        3000
 CUPERTINO       CA 93301           408 996-2323       7499       10000
 SANTA CLARA     CA 91003           408 376-9966       7844        5000
 PALO ALTO       CA 94301           415 364-9777       7521        6000
 SUNNYVALE       CA 93301           408 967-4398       7499       10000
 HIBBING         MN 55649           612 566-9123       7844        8000


 COMMENTS
 -----------------------------------------------------------------------------
 Very friendly people to work with -- sales rep likes to be called Mike.
 Rep called 5/8 about change in order - contact shipping.
 Company doing heavy promotion beginning 10/89. Prepare for large orders during orders during winter
 Contact rep about new line of tennis rackets.
 Customer with high market share (23%) due to aggressive advertising.
 Tends to order large amounts of merchandise at once. Accounting is considering raising their credit limit
 Support intensive. Orders small amounts (< 800) of merchandise at a time.
 First sporting goods store geared exclusively towards women. Unusual promotional style
```

PRICE Table

```
SQL> DESCRIBE price
```

```
Name                               Null?    Type
 ------------------------------- -------- ----
 PRODID                           NOT NULL NUMBER(6)
 STDPRICE                                  NUMBER(8,2)
 MINPRICE                                  NUMBER(8,2)
 STARTDATE                                 DATE
 ENDDATE                                   DATE
```

```
SQL> SELECT * FROM price;
```

```
   PRODID  STDPRICE  MINPRICE STARTDATE ENDDATE
--------- --------- --------- --------- ---------
   100871       4.8       3.2 01-JAN-85 01-DEC-85
   100890        58      46.4 01-JAN-85
   100890        54      40.5 01-JUN-84 31-MAY-84
   100860        35        28 01-JUN-86
   100860        32      25.6 01-JAN-86 31-MAY-86
   100860        30        24 01-JAN-85 31-DEC-85
   100861        45        36 01-JUN-86
   100861        42      33.6 01-JAN-86 31-MAY-86
   100861        39      31.2 01-JAN-85 31-DEC-85
   100870       2.8       2.4 01-JAN-86
   100870       2.4       1.9 01-JAN-85 01-DEC-85
   100871       5.6       4.8 01-JAN-86
   101860        24        18 15-FEB-85
   101863      12.5       9.4 15-FEB-85
   102130       3.4       2.8 18-AUG-85
   200376       2.4      1.75 15-NOV-86
   200380         4       3.2 15-NOV-86
```