6

Fine-Grained Access Control

ORACLE

Objectives

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

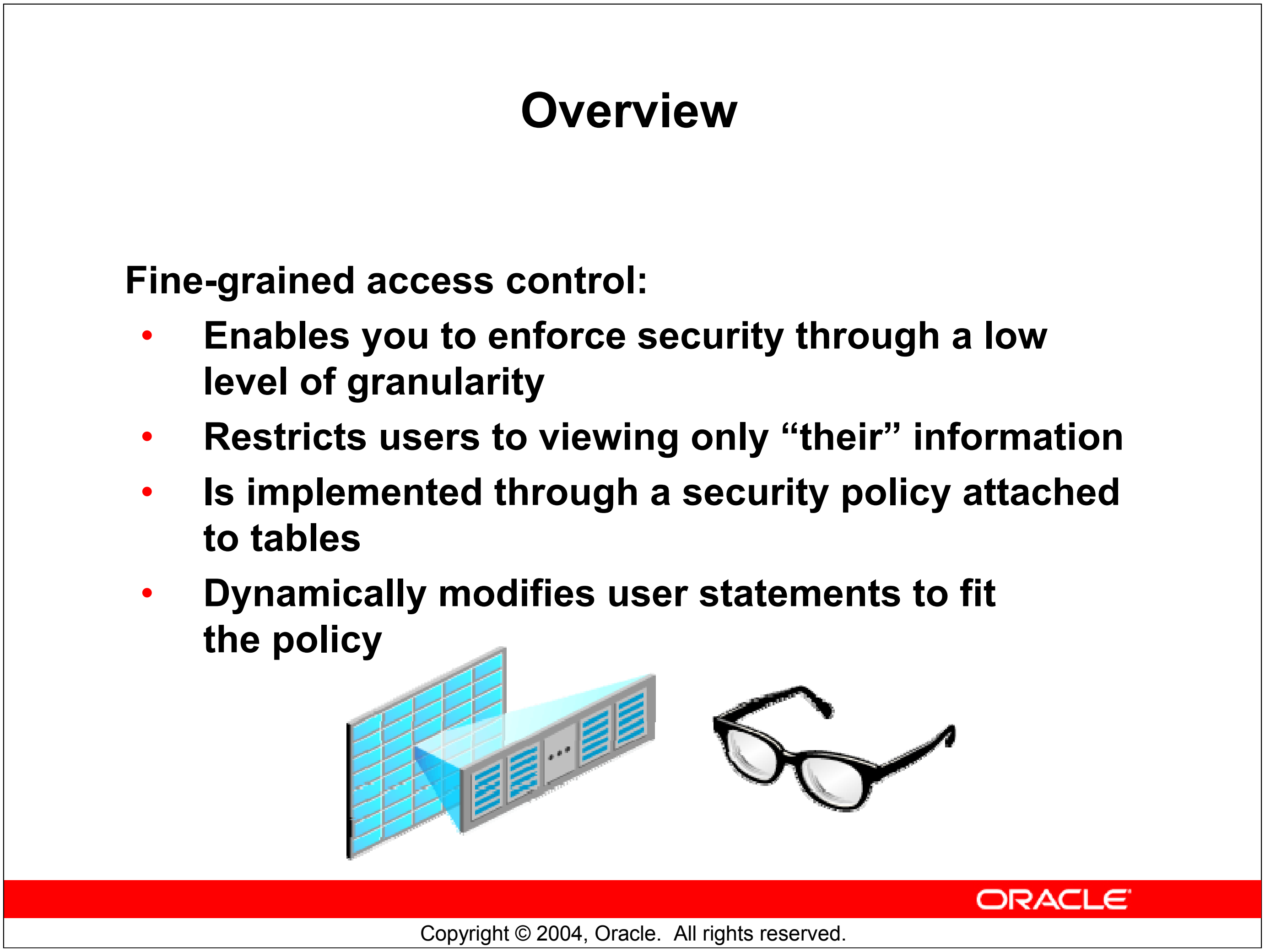
- **Describe the process of fine-grained access control**
- **Implement and test fine-grained access control**

ORACLE

Objectives

In this lesson, you will learn about the security features in the Oracle Database from an application developer's standpoint.

For more information about these features, refer to *Oracle Supplied PL/SQL Packages and Types Reference, Oracle Label Security Administrator's Guide, Oracle Single Sign-On Application Developer's Guide,* and *Oracle Security Overview*.



Overview

Fine-grained access control:

- Enables you to enforce security through a low level of granularity
- Restricts users to viewing only “their” information
- Is implemented through a security policy attached to tables
- Dynamically modifies user statements to fit the policy

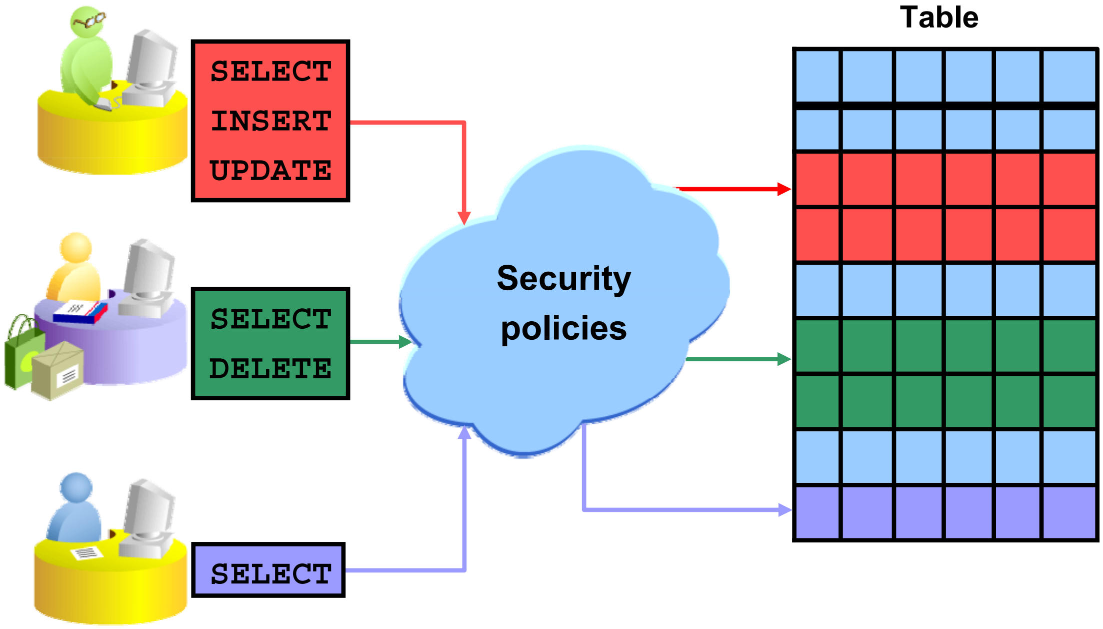
Fine-Grained Access Control

Fine-grained access control enables you to build applications that enforce security rules (or policies) at a low level of granularity. For example, you can use fine-grained access to restrict customers who access the Oracle server to see only their own account, physicians to see only the records of their own patients, or managers to see only the records of employees who work for them.

When you use fine-grained access control, you create security policy functions attached to the table or view on which you have based your application. Then, when a user enters a DML statement on that object, the Oracle server dynamically modifies the user’s statement—transparently to the user—so that the statement implements the correct access control.

Fine-grained access is also known as a virtual private database (VPD) because it implements row-level security, essentially giving the user access to his or her own private database. Fine-grained means at the individual row level.

Identifying Fine-Grained Access Features



Table

SELECT
INSERT
UPDATE

SELECT
DELETE

SELECT

Security policies

Features

You can use fine-grained access control to implement security rules called policies with functions, and then associate those security policies with tables or views. The database server automatically enforces those security policies, no matter how the data is accessed.

A security policy is a collection of rules needed to enforce the appropriate privacy and security rules into the database itself, making it transparent to users of the data structure.

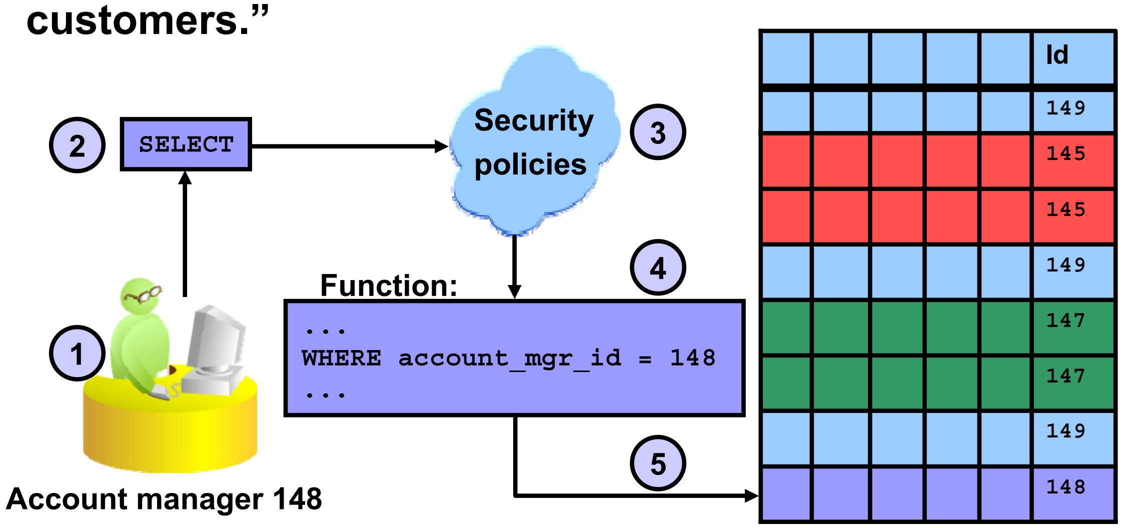
Attaching security policies to tables or views, rather than to applications, provides greater security, simplicity, and flexibility.

You can:

- Use different policies for `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements
- Use security policies only where you need them
- Use more than one policy for each table, including building on top of base policies in packaged applications
- Distinguish policies between different applications by using policy groups

How Fine-Grained Access Works

Implement the policy on the `CUSTOMERS` table: "Account managers can see only their own customers."

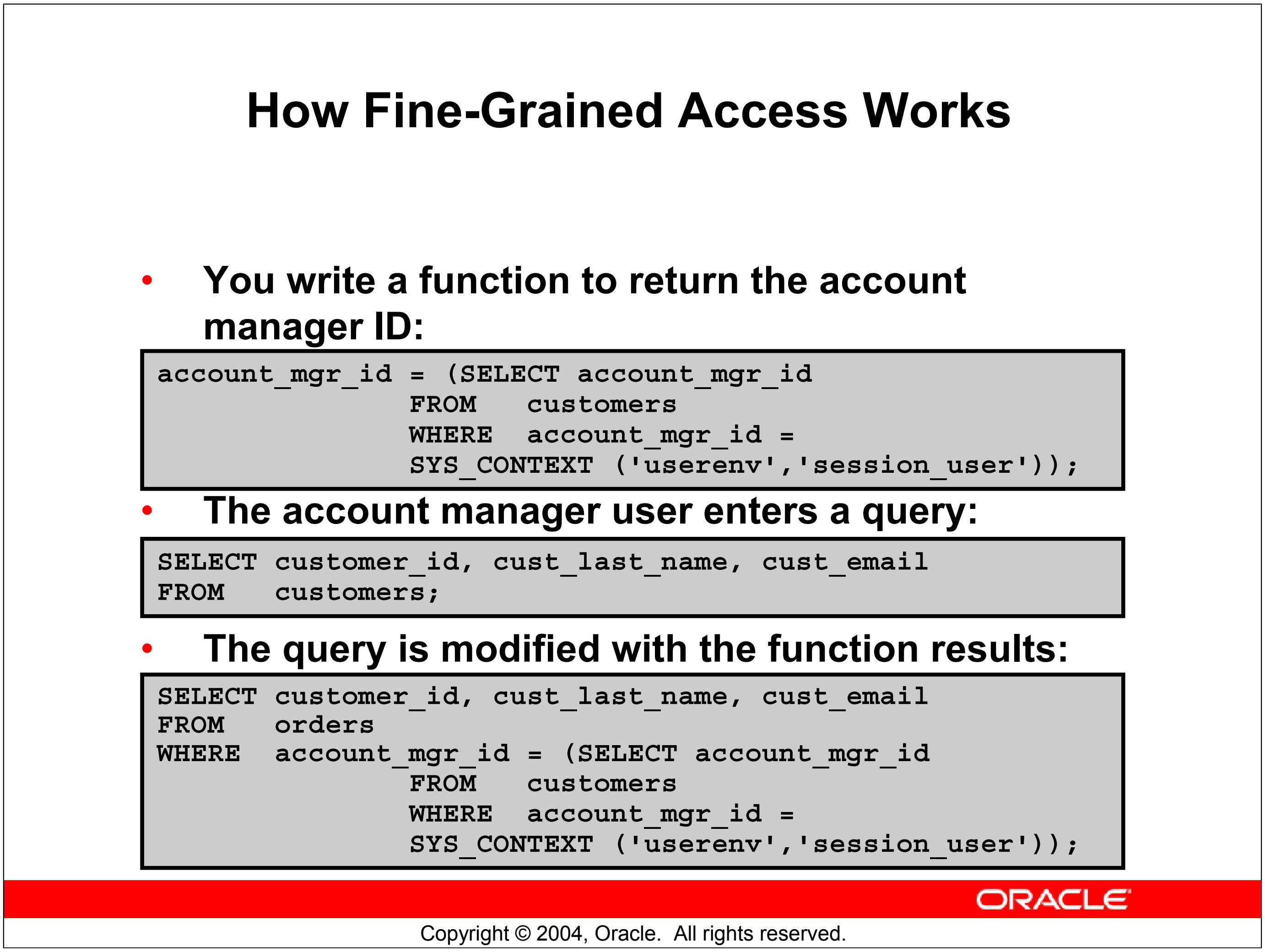


```
...
WHERE account_mgr_id = 148
...
```

How Fine-Grained Access Works

To implement a virtual private database so that each account manager can see only his or her own customers, you must do the following:

1. Create a function to add a `WHERE` clause identifying a selection criterion to a user's DML statement.
2. Have the user (the account manager) enter a DML statement.
3. Implement the security policy through the function you created. The Oracle server calls the function automatically.
4. Dynamically modify the user's statement through the function.
5. Execute the dynamically modified statement.



How Fine-Grained Access Works

- You write a function to return the account manager ID:

```
account_mgr_id = (SELECT account_mgr_id
                  FROM   customers
                  WHERE  account_mgr_id =
                  SYS_CONTEXT ('userenv','session_user'));
```

- The account manager user enters a query:

```
SELECT customer_id, cust_last_name, cust_email
FROM   customers;
```

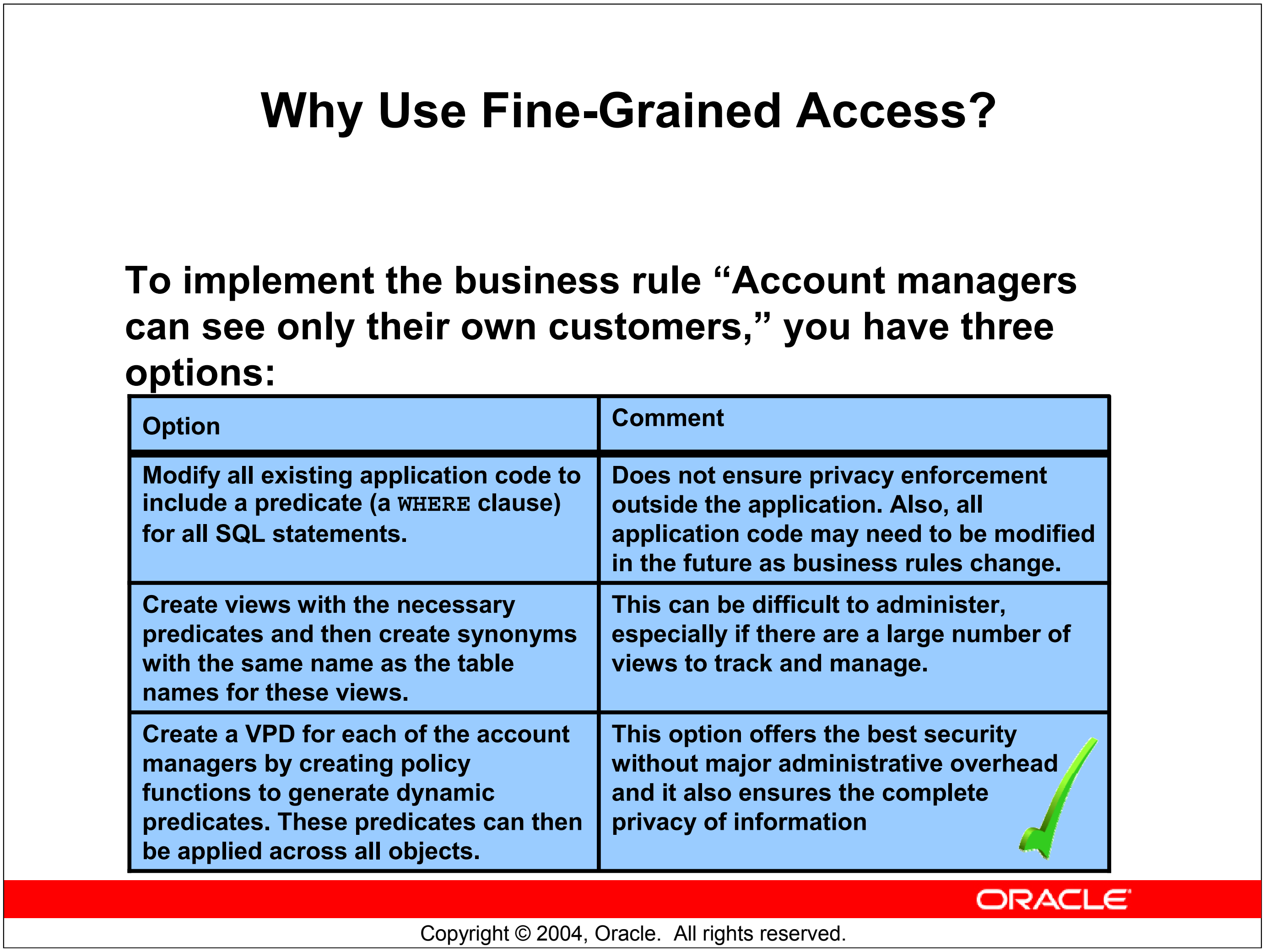
- The query is modified with the function results:

```
SELECT customer_id, cust_last_name, cust_email
FROM   orders
WHERE  account_mgr_id = (SELECT account_mgr_id
                         FROM   customers
                         WHERE  account_mgr_id =
                         SYS_CONTEXT ('userenv','session_user'));
```

How Fine-Grained Access Works (continued)

Fine-grained access control is based on a dynamically modified statement. In the example shown, the user enters a broad query against the CUSTOMERS table that retrieves customer names and e-mail names for a specific account manager. The Oracle server calls the function to implement the security policy. This modification is transparent to the user. It results in successfully restricting access to other customers' information, displaying only the information relevant to the account manager.

Note: SYS_CONTEXT is a function that returns a value for an attribute. This is explained in detail in a few pages.



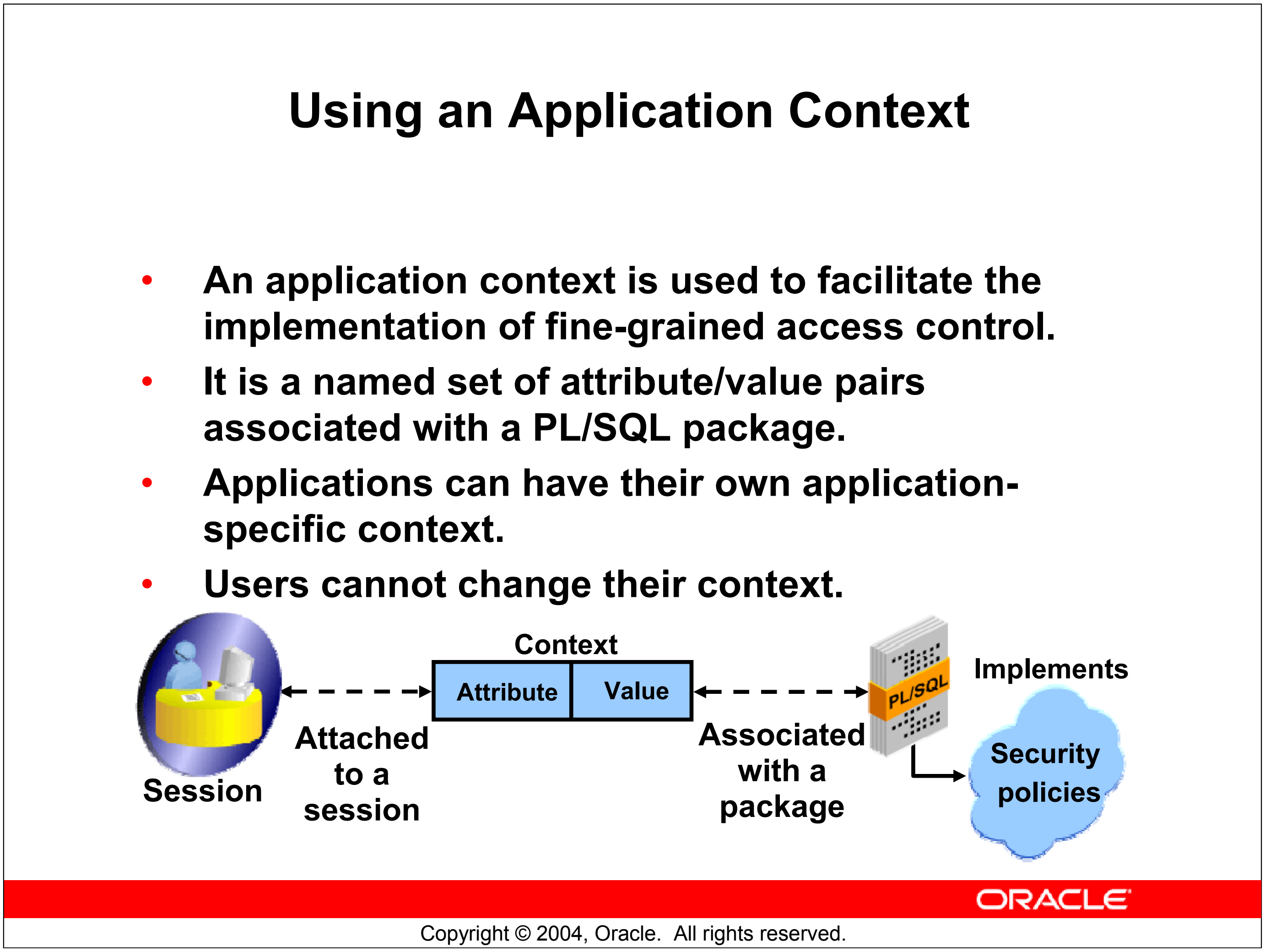
Why Use Fine-Grained Access?

To implement the business rule “Account managers can see only their own customers,” you have three options:

Option	Comment
Modify all existing application code to include a predicate (a `WHERE` clause) for all SQL statements.	Does not ensure privacy enforcement outside the application. Also, all application code may need to be modified in the future as business rules change.
Create views with the necessary predicates and then create synonyms with the same name as the table names for these views.	This can be difficult to administer, especially if there are a large number of views to track and manage.
Create a VPD for each of the account managers by creating policy functions to generate dynamic predicates. These predicates can then be applied across all objects.	This option offers the best security without major administrative overhead and it also ensures the complete privacy of information ✓

Why Choose Fine-Grained Access?

You can implement the business rule “Account managers can see only their own customers” through a few means. The options are listed above. By using fine-grained access, you have security implemented without a lot of overhead.



Using an Application Context

- An application context is used to facilitate the implementation of fine-grained access control.
- It is a named set of attribute/value pairs associated with a PL/SQL package.
- Applications can have their own application-specific context.
- Users cannot change their context.

What Is an Application Context?

An application context:

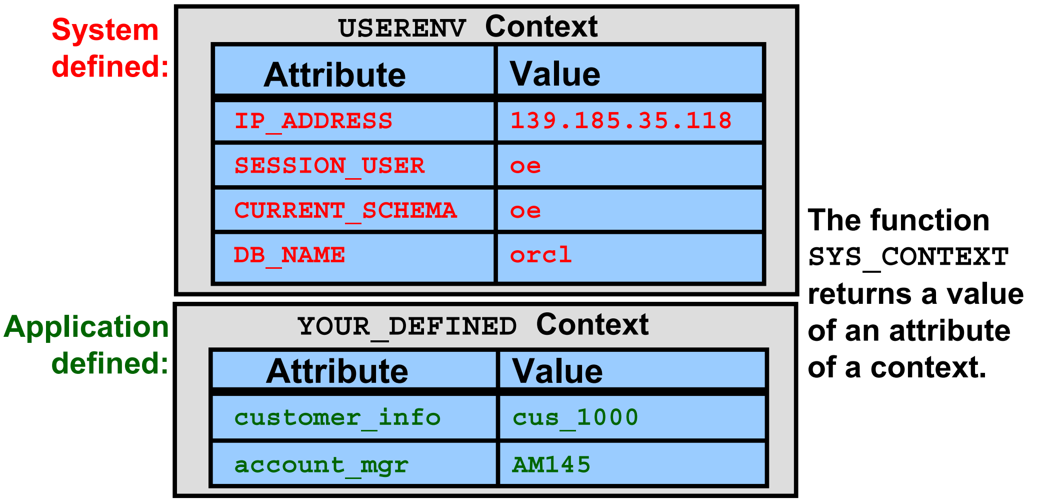
- Is a named set of attribute/value pairs associated with a PL/SQL package
- Is attached to a session
- Enables you to implement security policies with functions and then associate them with applications

A context is a named set of attribute/value pairs that are global to your session. You can define an application context, name it, and associate a value to that context with a PL/SQL package. Application context enables you to write applications that draw upon certain aspects of a user's session information. It provides a way to define, set, and access attributes that an application can use to enforce access control—specifically, fine-grained access control.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, you would limit the customers' to access their own orders (`ORDER_ID`) and customer number (`CUSTOMER_ID`). Or, you may limit an account manager (`ACCOUNT_MGR_ID`) to view only his or her customers. These values can be used as security attributes. Your application can use a context to set values that are accessed within your code and used to generate `WHERE` clause predicates for fine-grained access control.

An application context is owned by `SYS`.

Using an Application Context



System defined:

USERENV Context

Attribute	Value
IP_ADDRESS	139.185.35.118
SESSION_USER	oe
CURRENT_SCHEMA	oe
DB_NAME	orcl

Application defined:

YOUR_DEFINED Context

Attribute	Value
customer_info	cus_1000
account_mgr	AM145

The function SYS_CONTEXT returns a value of an attribute of a context.

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
------------------------------------------------------
 OE
```

Application Context

A predefined application context named USERENV is available to you. It has a predefined list of attributes. Predefined attributes can be very useful for access control. You find the values of the attributes in a context by using the SYS_CONTEXT function. Although the predefined attributes in the USERENV application context are accessed with the SYS_CONTEXT function, you cannot change them.

With the SYS_CONTEXT function, you pass the context name and the attribute name. The attribute value is returned.

The following statement returns the name of the database being accessed:

```
SELECT SYS_CONTEXT ('USERENV', 'DB_NAME')
FROM DUAL;

SYS_CONTEXT('USERENV','DB_NAME')
----------------------------------------------
ORCL
```

Creating an Application Context

```
CREATE [OR REPLACE] CONTEXT namespace
USING  [schema.]plsql_package
```

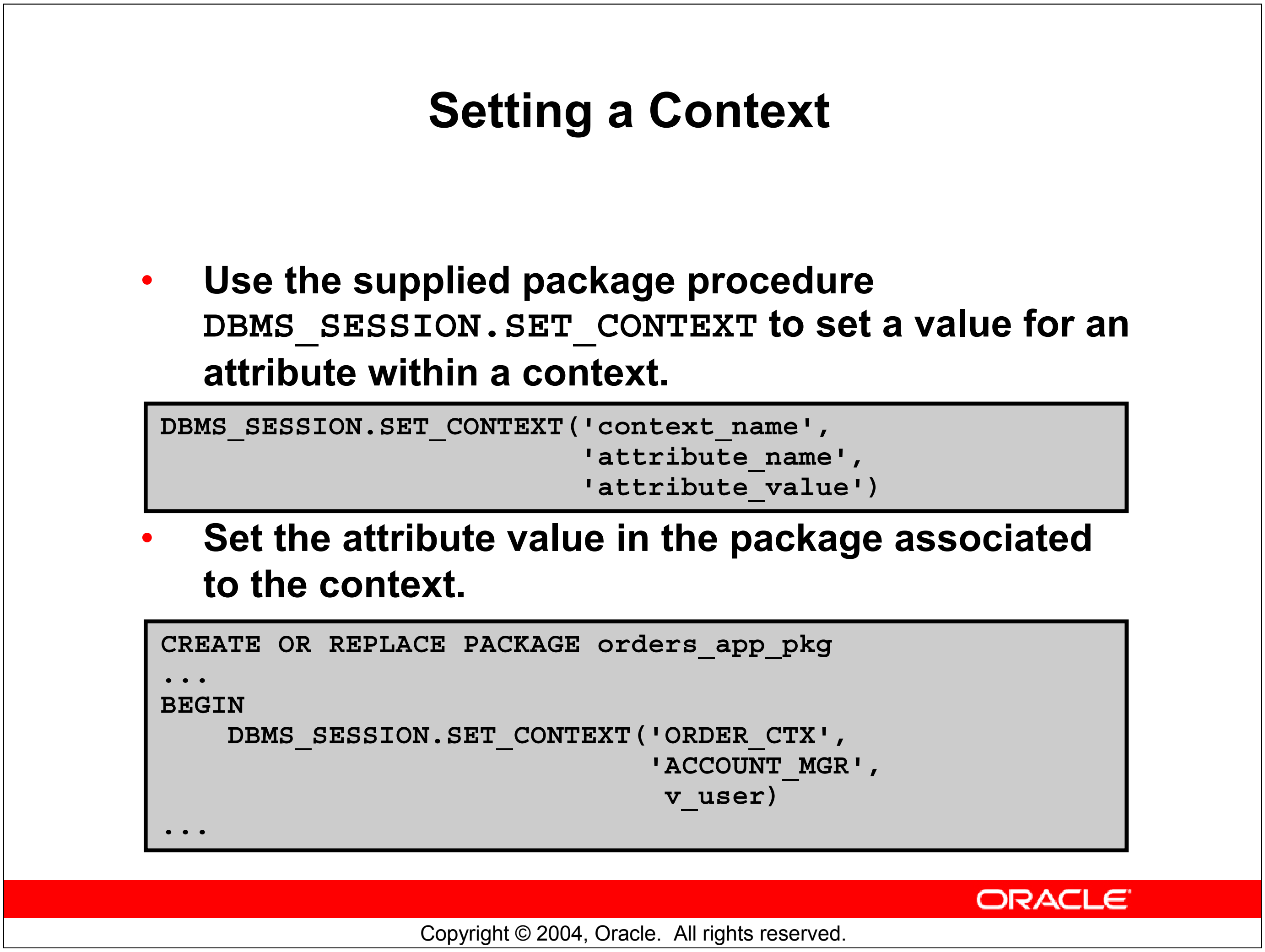
- Requires the `CREATE ANY CONTEXT` system privilege
- Parameters:
  - `namespace` is the name of the context.
  - `schema` is the name of the schema owning the PL/SQL package.
  - `plsql_package` is the name of the package used to set or modify the attributes of the context. (It does not need to exist at the time of the context creation.)

```
CREATE CONTEXT order_ctx USING oe.orders_app_pkg;

Context created.
```

Creating a Context

For fine-grained access where you want an account manager to view only his or her customers, customers can view their own information, and sales representatives can view only their own orders, you can create a context called `ORDER_CTX` and define for it the `ACCOUNT_MGR`, `CUST_ID`, and `SALE_REP` attributes.

Because a context is associated with a PL/SQL package, you need to name the package that you are tying to the context. This package does not need to exist at the time of context creation.



Setting a Context

- Use the supplied package procedure `DBMS_SESSION.SET_CONTEXT` to set a value for an attribute within a context.

```
DBMS_SESSION.SET_CONTEXT('context_name',
                         'attribute_name',
                         'attribute_value')
```

- Set the attribute value in the package associated to the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
...
BEGIN
    DBMS_SESSION.SET_CONTEXT('ORDER_CTX',
                             'ACCOUNT_MGR',
                              v_user)
...
```

Setting a Context

When a context is defined, you can use the DBMS_SESSION.SET_CONTEXT procedure to set a value for an attribute within a context. The attribute is set in the package associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
 PROCEDURE set_app_context;
END;
/
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
 c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
 PROCEDURE set_app_context
 IS
    v_user VARCHAR2(30);
 BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_SESSION.SET_CONTEXT
    (c_context, 'ACCOUNT_MGR', v_user);
 END;
END;
/
```

Setting a Context (continued)

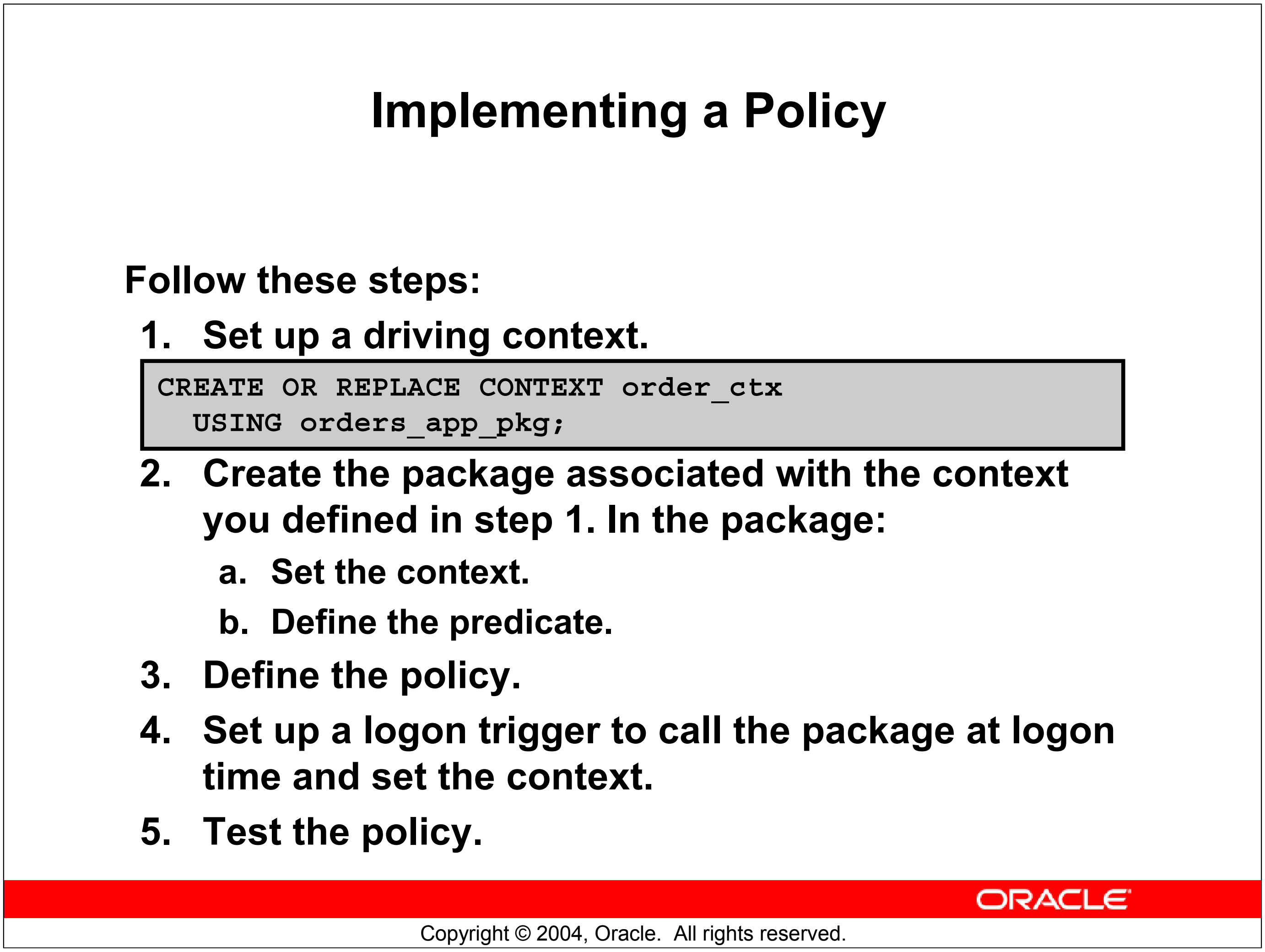
In the example on the previous page, the context `ORDER_CTX` has the `ACCOUNT_MGR` attribute set to the current user logged (determined by the `USER` function).

For this example, assume that users `AM145`, `AM147`, `AM148`, and `AM149` exist. As each user logs on and the `DBMS_SESSION.SET_CONTEXT` is invoked, the attribute value for that `ACCOUNT_MGR` is set to the user ID.

```
GRANT EXECUTE ON oe.orders_app_pkg
  TO AM145, AM147, AM148, AM149;

CONNECT AM145/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
---------------------------------------------------------
AM145
```

If you switch the user ID, the attribute value is also changed to reflect the current user.

```
CONNECT AM147/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
---------------------------------------------------------
AM147
```



Implementing a Policy

Follow these steps:

1. Set up a driving context.

```
CREATE OR REPLACE CONTEXT order_ctx
  USING orders_app_pkg;
```

2. Create the package associated with the context you defined in step 1. In the package:
   a. Set the context.
   b. Define the predicate.
3. Define the policy.
4. Set up a logon trigger to call the package at logon time and set the context.
5. Test the policy.

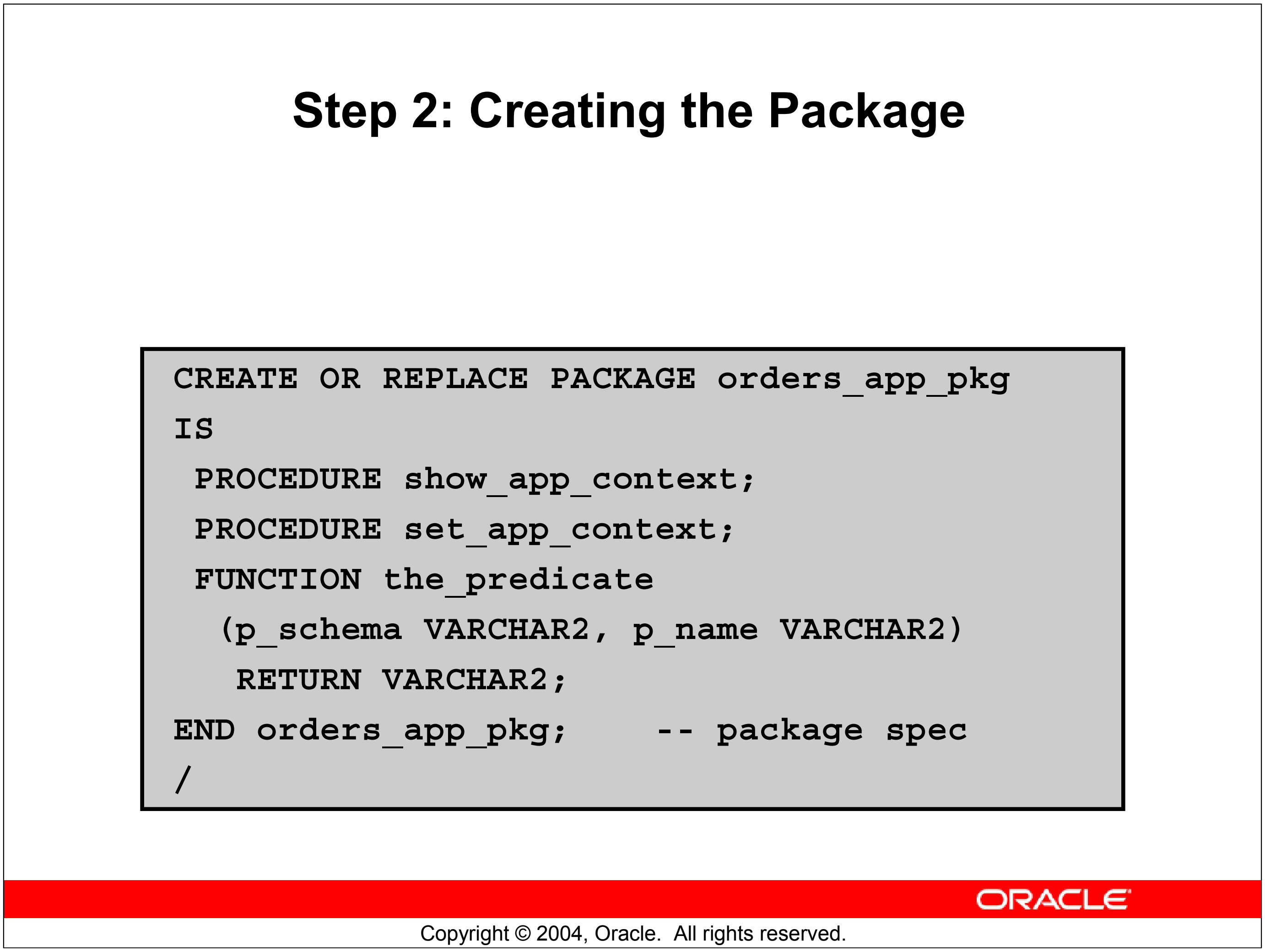
Implementing a Policy

In this example, assume that the users AM145, AM147, AM148, and AM149 exist. Next, a context and a package associated with the context is created. The package will be owned by OE.

Step 1: Set Up a Driving Context

Use the CREATE CONTEXT syntax to create a context.

```
CONNECT /AS sysdba

CREATE CONTEXT order_ctx USING oe.orders_app_pkg;
```



Step 2: Creating the Package

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
 PROCEDURE show_app_context;
 PROCEDURE set_app_context;
 FUNCTION the_predicate
   (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END orders_app_pkg;    -- package spec
/
```

Implementing a Policy (continued)

Step 2: Create a Package

In the OE schema, the ORDERS_APP_PKG is created. This package contains three routines:

- **show_app_context:** For learning and testing purposes, this procedure will display a context attribute and value.
- **set_app_context:** This procedure sets a context attribute to a specific value.
- **the_predicate:** This function builds the predicate (the WHERE clause) that will control the rows visible in the CUSTOMERS table to a user. (Note that this function requires two input parameters. An error will occur when the policy is implemented if you exclude these two parameters.)

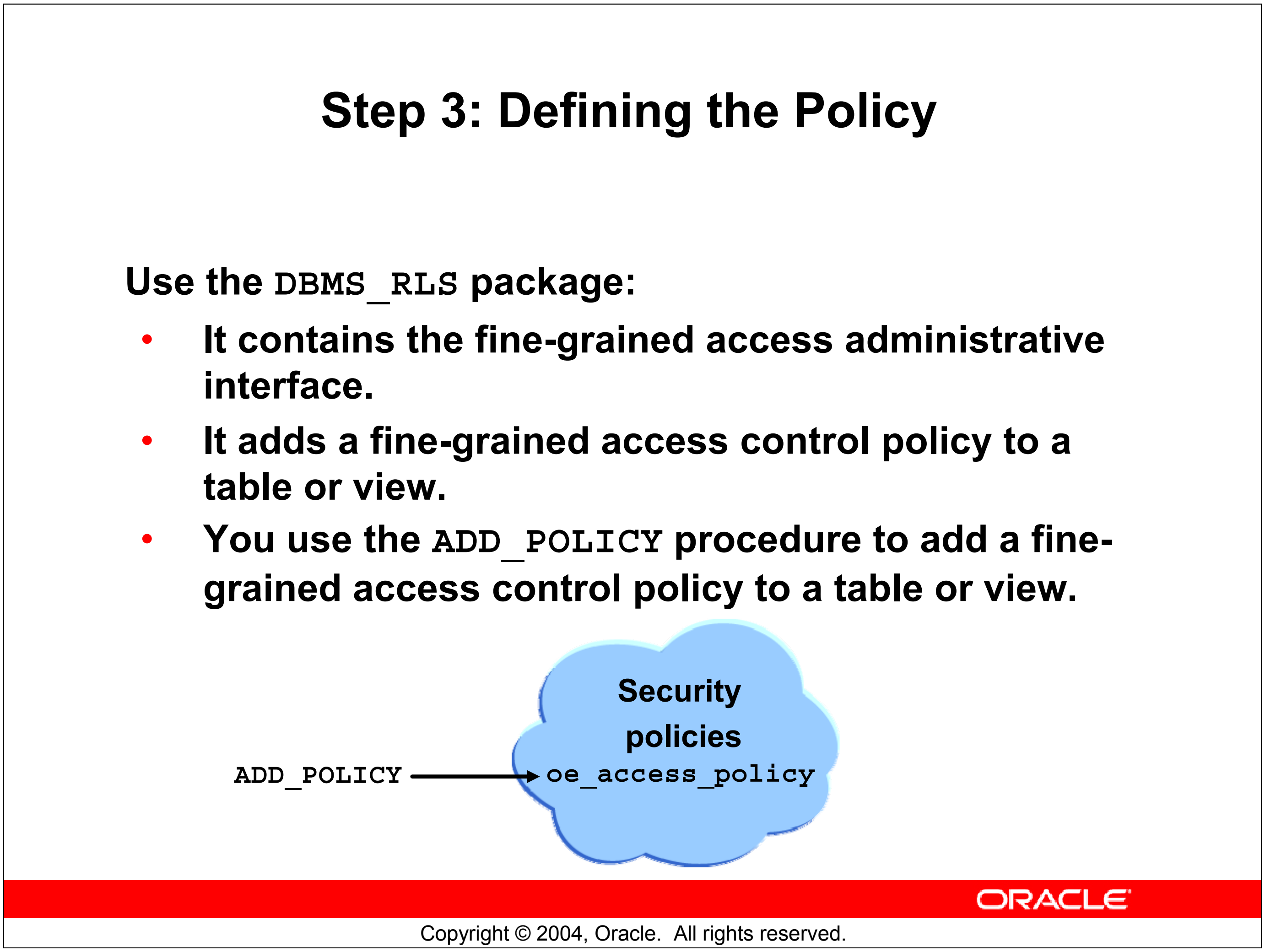
Implementing a Policy (continued)

Step 2: Create a Package (continued)

```
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
  c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
  c_attrib  CONSTANT VARCHAR2(30) := 'ACCOUNT_MGR';

PROCEDURE show_app_context
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Type: ' || c_attrib ||
   ' - ' || SYS_CONTEXT(c_context, c_attrib));
END show_app_context;

PROCEDURE set_app_context
  IS
    v_user VARCHAR2(30);
BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_SESSION.SET_CONTEXT
    (c_context, c_attrib, v_user);
END set_app_context;

FUNCTION the_predicate
(p_schema VARCHAR2, p_name VARCHAR2)
RETURN VARCHAR2
IS
  v_context_value VARCHAR2(100) :=
     SYS_CONTEXT(c_context, c_attrib);
  v_restriction VARCHAR2(2000);
BEGIN
  IF v_context_value LIKE 'AM%'  THEN
    v_restriction :=
     'ACCOUNT_MGR_ID =
      SUBSTR(''' || v_context_value || ''', 3, 3)';
  ELSE
    v_restriction := null;
  END IF;
  RETURN v_restriction;
END the_predicate;

END orders_app_pkg; -- package body
/
```

Note that the THE_PREDICATE function builds the WHERE clause and stores it in the V_RESTRICTION variable. If the SYS_CONTEXT function returns an attribute value that starts with AM, then the WHERE clause is built with ACCOUNT_MGR_ID = *the last three characters of the attribute value*. If the user is AM145, then the WHERE clause will be:

```
WHERE account_mgr_id = 145
```



Step 3: Defining the Policy

Use the DBMS_RLS package:

- It contains the fine-grained access administrative interface.
- It adds a fine-grained access control policy to a table or view.
- You use the ADD_POLICY procedure to add a fine-grained access control policy to a table or view.

Security policies
ADD_POLICY
oe_access_policy

Implementing a Policy (continued)

The DBMS_RLS package contains the fine-grained access control administrative interface. The package holds several procedures. To add a policy, you use the ADD_POLICY procedure within the DBMS_RLS package.

Note: DBMS_RLS is available with the Enterprise Edition only.

Step 3: Define the Policy

The DBMS_RLS.ADD_POLICY procedure adds a fine-grained access control policy to a table or view. The procedure causes the current transaction, if any, to commit before the operation is carried out. However, this does not cause a commit first if it is inside a DDL event trigger. These are the parameters for the ADD_POLICY procedure:

```
DBMS_RLS.ADD_POLICY (
  object_schema   IN VARCHAR2 := NULL,
  object_name     IN VARCHAR2,
  policy_name     IN VARCHAR2,
  function_schema IN VARCHAR2 := NULL,
  policy_function IN VARCHAR2,
  statement_types IN VARCHAR2 := NULL,
  update_check    IN BOOLEAN := FALSE,
  enable          IN BOOLEAN := TRUE);
```

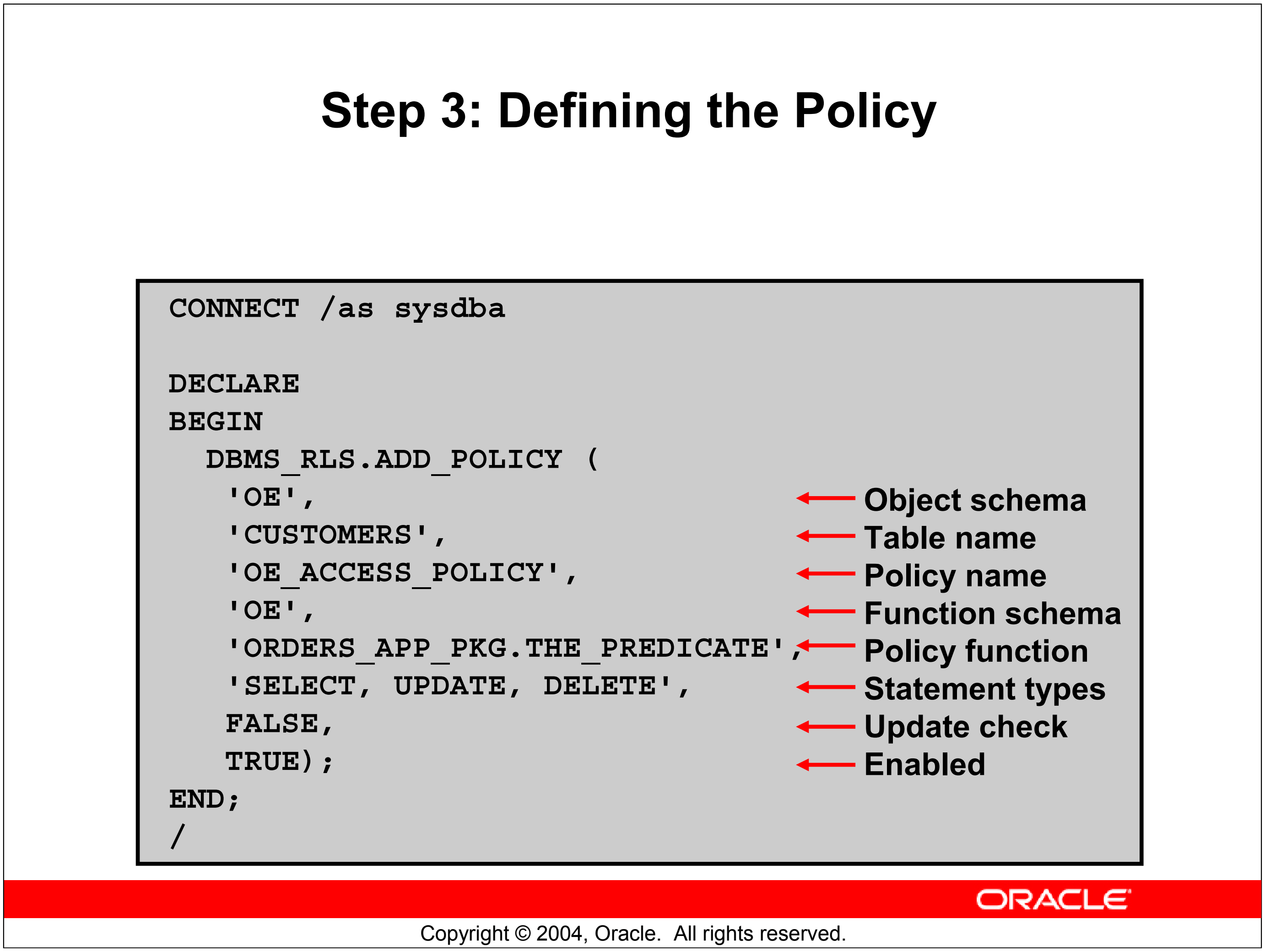
Implementing a Policy (continued)

Step 3: Define the Policy (continued)

Parameter	Description
OBJECT_SCHEMA	Schema containing the table or view (logon user, if NULL)
OBJECT_NAME	Name of table or view to which the policy is added
POLICY_NAME	Name of policy to be added. It must be unique for the same table or view.
FUNCTION_SCHEMA	Schema of the policy function (logon user, if NULL)
POLICY_FUNCTION	Name of a function that generates a predicate for the policy. If the function is defined within a package, then the name of the package must be present.
STATEMENT_TYPES	Statement types that the policy will apply. It can be any combination of SELECT, INSERT, UPDATE, and DELETE. The default is to apply to all these types.
UPDATE_CHECK	Optional argument for the INSERT or UPDATE statement types. The default is FALSE. Setting update_check to TRUE causes the server to also check the policy against the value after insert or update.
ENABLE	Indicates if the policy is enabled when it is added. The default is TRUE.

Below is a list of the procedures contained in the DBMS_RLS package. For detailed information, refer to the *PL/SQL Packages and Types Reference 10g Release 1 (10.1)* reference manual.

Procedure	Description
ADD_POLICY	Adds a fine-grained access control policy to a table or view
DROP_POLICY	Drops a fine-grained access control policy from a table or view
REFRESH_POLICY	Causes all the cached statements associated with the policy to be reparsed
ENABLE_POLICY	Enables or disables a fine-grained access control policy
CREATE_POLICY_GROUP	Creates a policy group
ADD_GROUPED_POLICY	Adds a policy associated with a policy group
ADD_POLICY_CONTEXT	Adds the context for the active application
DELETE_POLICY_GROUP	Deletes a policy group
DROP_GROUPED_POLICY	Drops a policy associated with a policy group
DROP_POLICY_CONTEXT	Drops a driving context from the object so that it will have one less driving context
ENABLE_GROUPED_POLICY	Enables or disables a row-level group security policy
REFRESH_GROUPED_POLICY	Reparses the SQL statements associated with a refreshed policy



Step 3: Defining the Policy

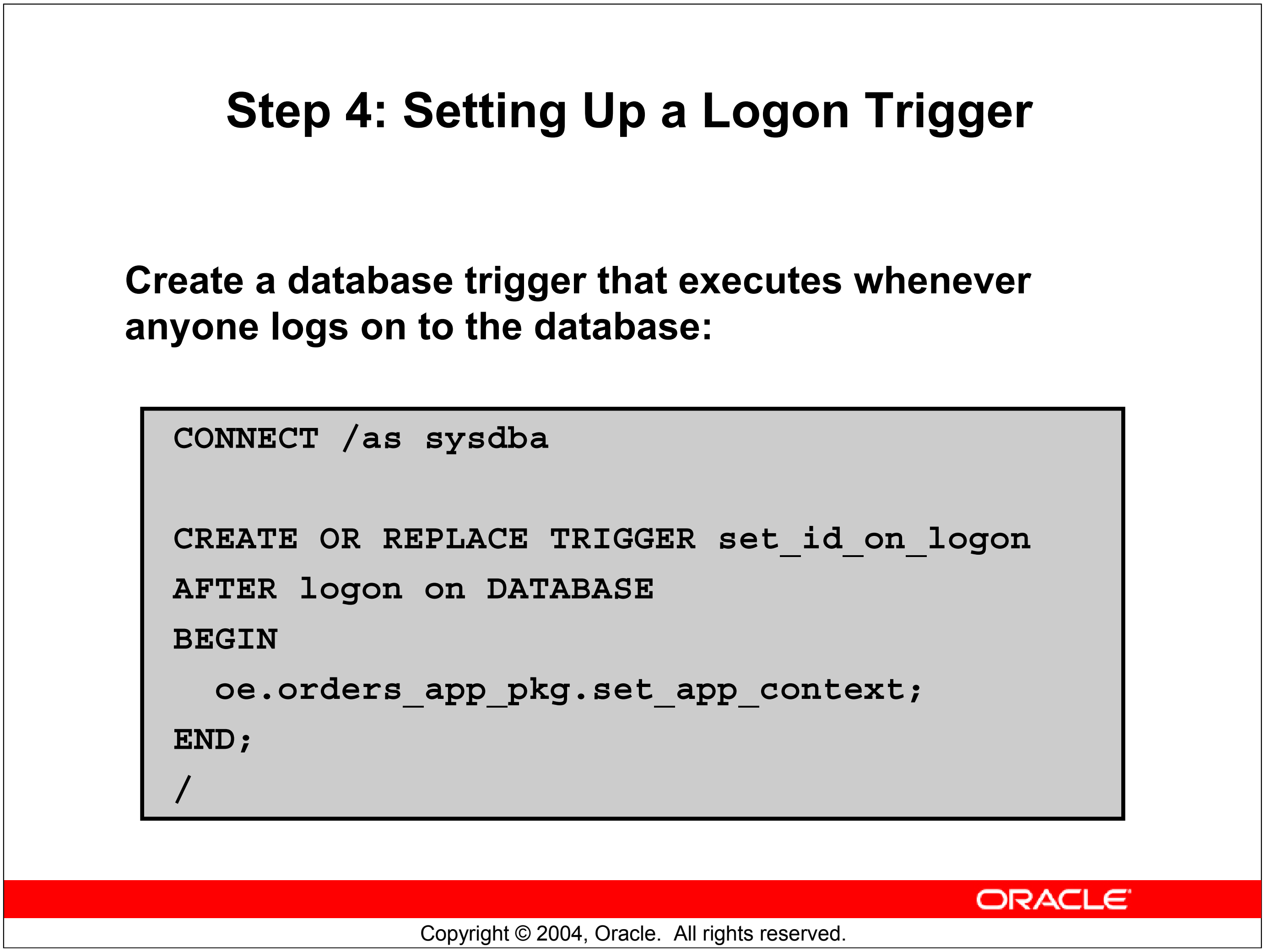
```
CONNECT /as sysdba

DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',                                ← Object schema
    'CUSTOMERS',                         ← Table name
    'OE_ACCESS_POLICY',                  ← Policy name
    'OE',                                ← Function schema
    'ORDERS_APP_PKG.THE_PREDICATE',      ← Policy function
    'SELECT, UPDATE, DELETE',            ← Statement types
    FALSE,                               ← Update check
    TRUE);                               ← Enabled
END;
/
```

Implementing a Policy (continued)

Step 3: Define the Policy (continued)

The security policy OE_ACCESS_POLICY is created and added with the DBMS_RLS.ADD_POLICY procedure. The predicate function that defines how the policy is to be implemented is associated with the policy being added.

This example specifies that whenever a SELECT, UPDATE, or DELETE statement on the OE.CUSTOMERS table is executed, the predicate function return result is appended to the end of the WHERE clause.



Step 4: Setting Up a Logon Trigger

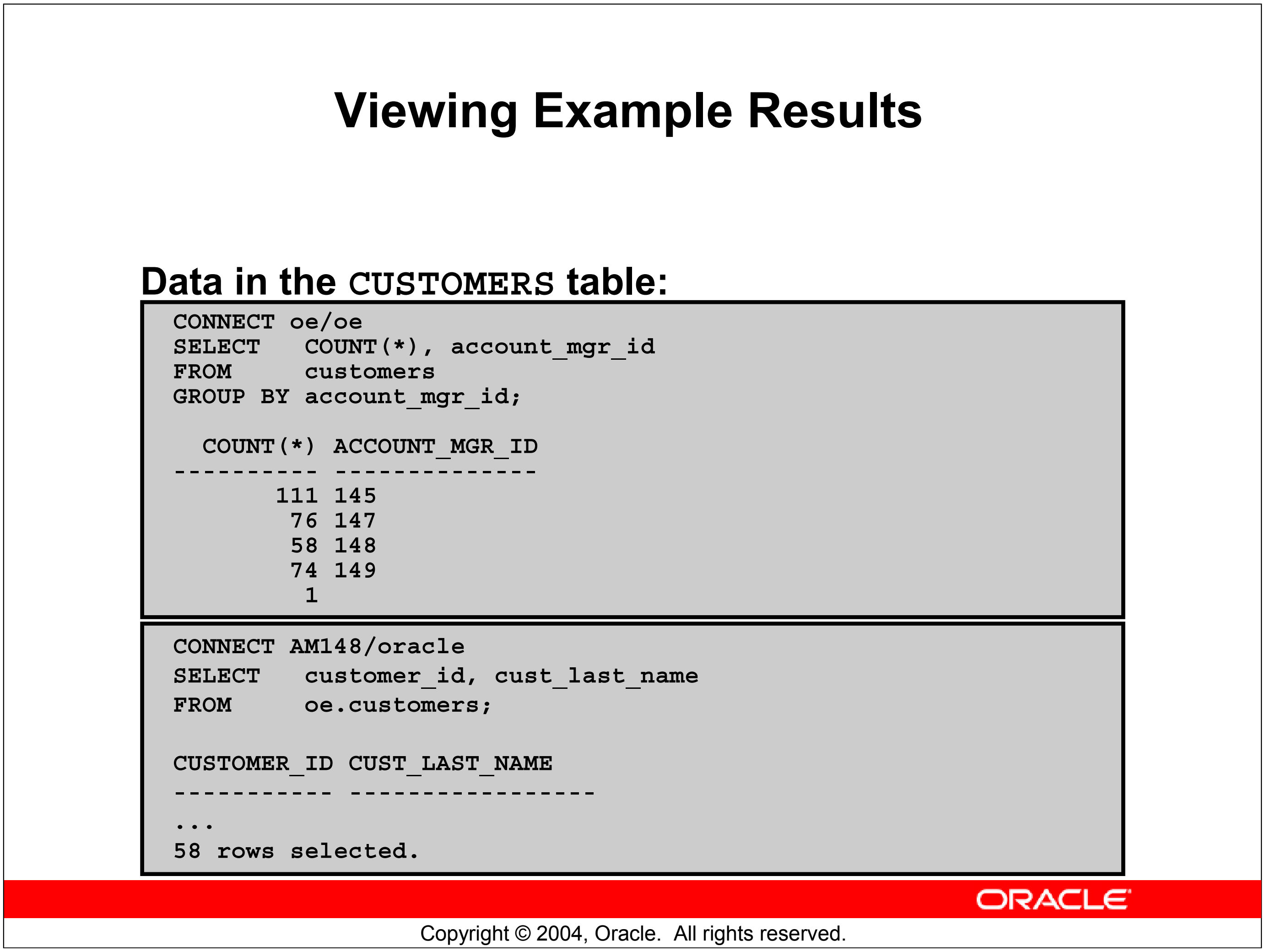
Create a database trigger that executes whenever anyone logs on to the database:

```
CONNECT /as sysdba

CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
  oe.orders_app_pkg.set_app_context;
END;
/
```

Implementing a Policy (continued)

Step 4: Set Up a Logon Trigger

After the context is created, the security package is defined, the predicate is defined, and the policy is defined, you need to create a logon trigger to implement fine-grained access control. This trigger causes the context to be set as each user is logged on.



Viewing Example Results

Data in the `CUSTOMERS` table:

```
CONNECT oe/oe
SELECT   COUNT(*), account_mgr_id
FROM     customers
GROUP BY account_mgr_id;

  COUNT(*) ACCOUNT_MGR_ID
---------- --------------
       111 145
        76 147
        58 148
        74 149
         1
```

```
CONNECT AM148/oracle
SELECT   customer_id, cust_last_name
FROM     oe.customers;

CUSTOMER_ID CUST_LAST_NAME
----------- -----------------
...
58 rows selected.
```

Example Results

The AM148 user who logs on will see only the rows in the CUSTOMERS table that are defined by the predicate function. The user can issue SELECT, UPDATE and DELETE statements against the CUSTOMERS table, but only the rows defined by the predicate function can be manipulated.

```
UPDATE oe.customers
SET credit_limit = credit_limit + 5000
WHERE customer_id = 101;

0 rows updated.
```

The AM148 user does not have access to customer ID 101. Customer ID 101 has the account manager of 145. To user AM148, any updates, deletes, or selects attempted on customers that do not have him as an account manager are not performed. It is as though those customers do not exist.

Using Data Dictionary Views

- `USER_POLICIES`
- `ALL_POLICIES`
- `DBA_POLICIES`
- `ALL_CONTEXT`
- `DBA_CONTEXT`

Data Dictionary Views

You can query the data dictionary views to find out information about the policies available in your schema.

View	Description
`USER_POLICIES`	All policies owned by the current schema
`ALL_POLICIES`	All policies owned or accessible by the current schema
`DBA_POLICIES`	All policies
`ALL_CONTEXT`	All active context namespaces defined in the session
`DBA_CONTEXT`	All context namespace information (active and inactive)

Using the ALL_CONTEXT Dictionary View

Use ALL_CONTEXT to see the active context namespaces defined in your session:

```
CONNECT AM148/oracle

SELECT *
FROM   all_context;

NAMESPACE        SCHEMA     PACKAGE
---------------- ---------  ----------
ORDER_CTX        OE         ORDERS_APP_PKG
```

ORACLE

Dictionary Views

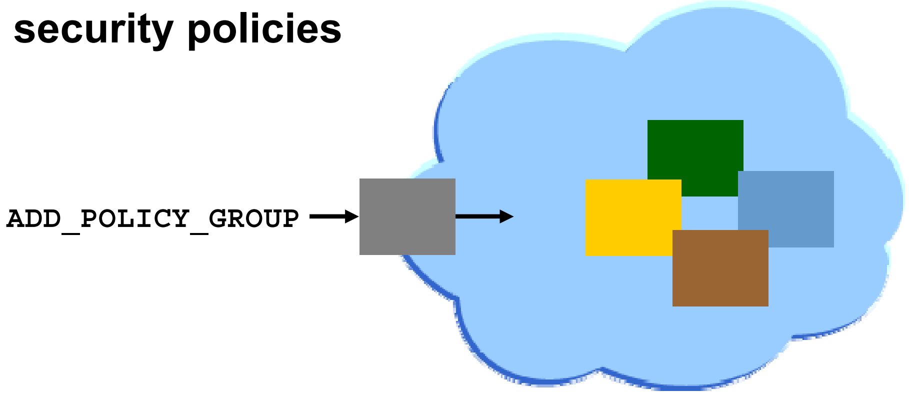
You can use the ALL_CONTEXT dictionary view to view information about contexts to which you have access. In the slide, the NAMESPACE column is equivalent to the context name.

You can use the ALL_POLICIES dictionary view to view information about polices to which you have access. In the example below, information is shown on the OE_ACCESS_POLICY policy.

```
SELECT object_name, policy_name, pf_owner, package,
       function, sel, ins, upd, del
FROM all_policies;

OBJECT_NAME                    POLICY_NAME
------------------------------ ------------------------------
PF_OWNER                       PACKAGE
------------------------------ ------------------------------
FUNCTION                       SEL INS UPD DEL
------------------------------ --- --- --- ---
CUSTOMERS                      OE_ACCESS_POLICY
OE                             ORDERS_APP_PKG
THE_PREDICATE                  YES NO  YES YES
```

Policy Groups

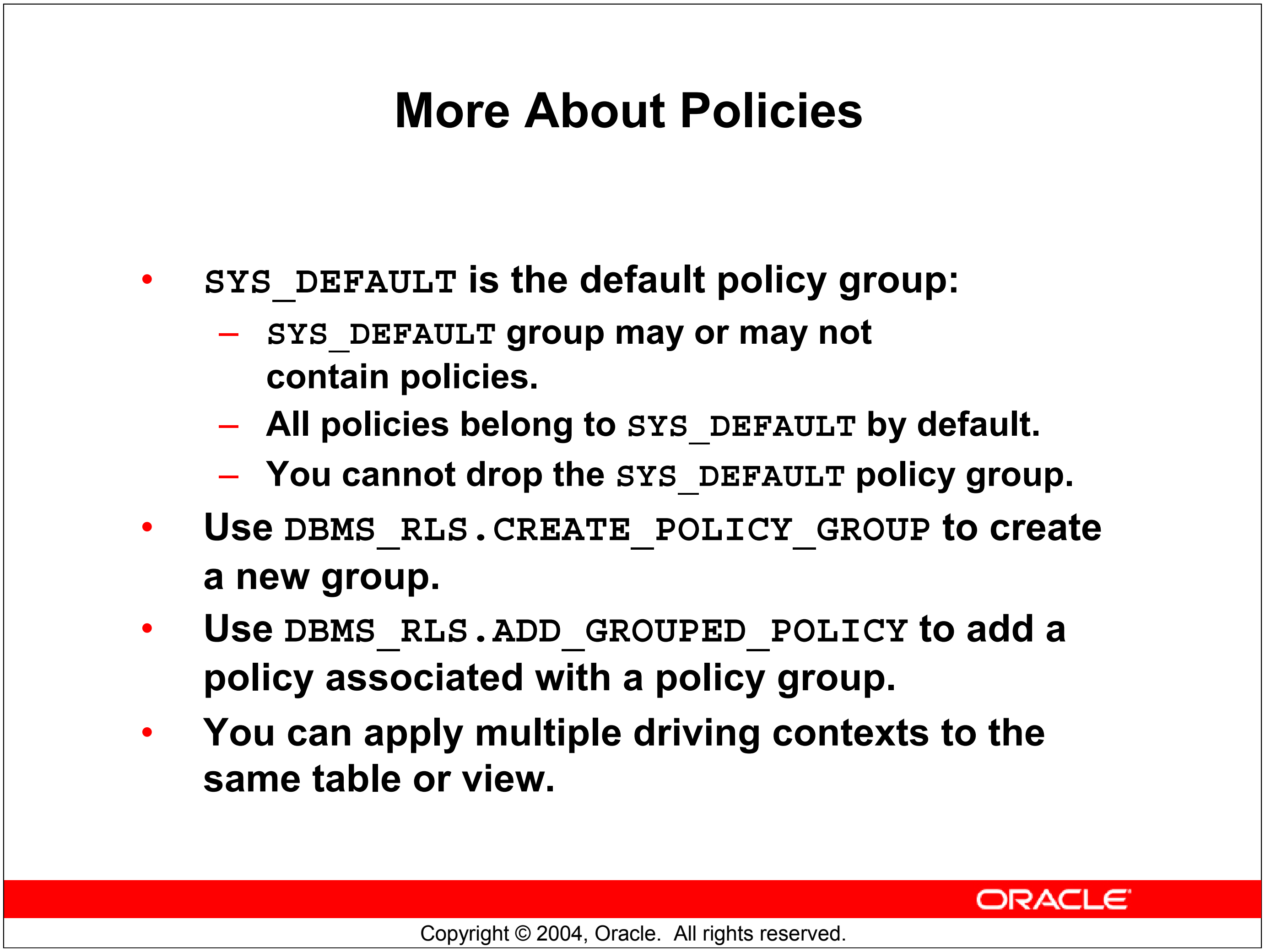
- Indicate a set of policies that belong to an application
- Are set up by a DBA through an application context, called a driving context
- Use the `DBMS_RLS` package to administer the security policies



Policy Groups

Policy groups were introduced in Oracle9*i*, release 1 (9.0.1). The database administrator designates an application context, called a driving context, to indicate the policy group in effect. When tables or views are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all the associated policies that belong to that policy group.

The PL/SQL `DBMS_RLS` package enables you to administer your security policies and groups. Using this package, you can add, drop, enable, disable, and refresh the policy groups you create.



More About Policies

- `SYS_DEFAULT` is the default policy group:
  - `SYS_DEFAULT` group may or may not contain policies.
  - All policies belong to `SYS_DEFAULT` by default.
  - You cannot drop the `SYS_DEFAULT` policy group.
- Use `DBMS_RLS.CREATE_POLICY_GROUP` to create a new group.
- Use `DBMS_RLS.ADD_GROUPED_POLICY` to add a policy associated with a policy group.
- You can apply multiple driving contexts to the same table or view.

More About Policies

A policy group is a set of security policies that belong to an application. You can designate an application context (known as a driving context) to indicate the policy group in effect. When the tables or views are accessed, the server looks up the driving context (that is also known as policy context) to determine the policy group in effect. It enforces all the associated policies that belong to that policy group.

By default, all policies belong to the SYS_DEFAULT policy group. Policies defined in this group for a particular table or view will always be executed along with the policy group specified by the driving context. The SYS_DEFAULT policy group may or may not contain policies. If you attempt to drop the SYS_DEFAULT policy group, an error will be raised. If you add policies associated with two or more objects to the SYS_DEFAULT policy group, then each such object will have a separate SYS_DEFAULT policy group associated with it. For example, the CUSTOMERS table in the OE schema has one SYS_DEFAULT policy group, and the ORDERS table in the OE schema has a different SYS_DEFAULT policy group associated with it. If you add policies associated with two or more objects, then each such object will have a separate SYS_DEFAULT policy group associated with it.

```
SYS_DEFAULT
 - policy1 (OE/CUSTOMERS)
 - policy3 (OE/CUSTOMERS)
SYS_DEFAULT
 - policy2 (OE/ORDERS)
```

More About Policies (continued)

When adding the policy to a table or view, you can use the `DBMS_RLS.ADD_GROUPED_POLICY` interface to specify the group to which the policy belongs. To specify which policies will be effective, you add a driving context using the `DBMS_RLS.ADD_POLICY_CONTEXT` interface. If the driving context returns an unknown policy group, an error is returned.

If the driving context is not defined, then all policies are executed. Likewise, if the driving context is `NULL`, then policies from all policy groups are enforced. In this way, an application that accesses the data cannot bypass the security setup module (that sets up application context) to avoid any applicable policies.

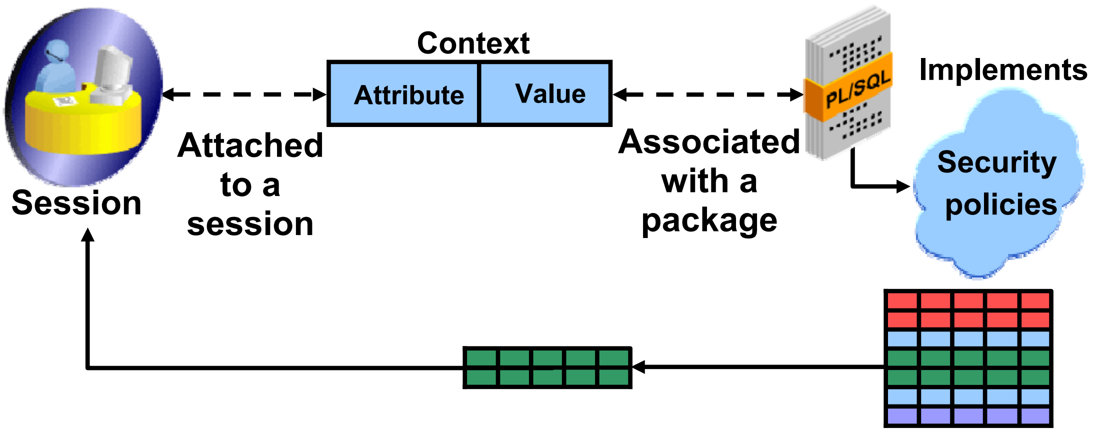
You can apply multiple driving contexts to the same table or view, and each of them will be processed individually. In this way, you can configure multiple active sets of policies to be enforced.

You can create a new policy using the `DBMS_RLS` package either from the command line or programmatically, or access the Oracle Policy Manager graphical user interface in Oracle Enterprise Manager.

Summary

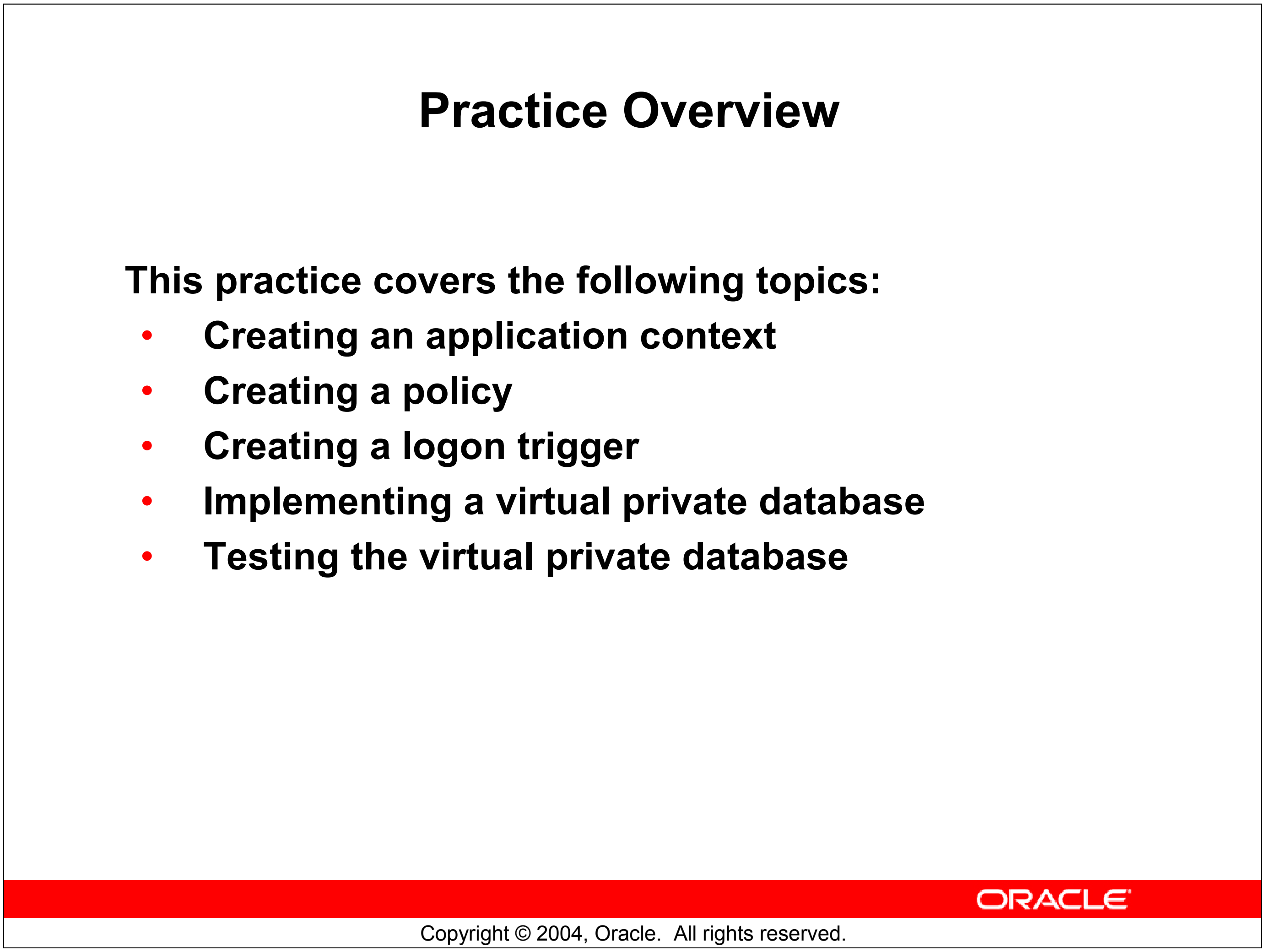
In this lesson, you should have learned how to:

- Describe the process of fine-grained access control
- Implement and test fine-grained access control



Summary

In this lesson you should have learned about fine-grained access control and the steps required to implement a virtual private database.



Practice Overview

This practice covers the following topics:

- Creating an application context
- Creating a policy
- Creating a logon trigger
- Implementing a virtual private database
- Testing the virtual private database

Practice Overview

In this practice you will implement and test fine-grained access control.

Practice 6

In this practice you will define an application context and security policy to implement the policy: “Sales Representatives can see their own order information only in the ORDERS table.” You will create sales representative IDs to test the success of your implementation.

Examine the definition of the ORDERS table, and the sales representative’s data:

```
DESCRIBE orders
Name               Null?    Type
------------------ -------- --------------------------------
ORDER_ID           NOT NULL NUMBER(12)
ORDER_DATE         NOT NULL TIMESTAMP(6) WITH LOCAL TIME ZONE
ORDER_MODE                  VARCHAR2(8)
CUSTOMER_ID        NOT NULL NUMBER(6)
ORDER_STATUS                NUMBER(2)
ORDER_TOTAL                 NUMBER(8,2)
SALES_REP_ID                NUMBER(6)
PROMOTION_ID                NUMBER(6)

SELECT sales_rep_id, count(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5
         154         10
         155          5
         156          5
         158          7
         159          7
         160          6
         161         13
         163         12
                     35
10 rows selected.
```

1. Examine, and then run the lab_06_01.sql script.
   This script will create the sales representative’s ID accounts with appropriate privileges to access the database.

2. Set up an application context:

   Connect to the database as SYSDBA before creating this context.

   Create an application context named sales_orders_ctx.

   Associate this context to the oe.sales_orders_pkg.

Practice 6 (continued)

3. Connect as OE/OE.

   Examine this package specification:

```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
 PROCEDURE set_app_context;
 FUNCTION the_predicate
  (p_schema VARCHAR2, p_name VARCHAR2)
   RETURN VARCHAR2;
END sales_orders_pkg;    -- package spec
/
```

   Create this package specification and then the package body in the OE schema.

   When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30) := 'SALES_ORDER_CTX';
c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';
```

Use these constants in the SET_APP_CONTEXT procedure to set the application context to the current user.

4. Connect as SYSDBA and define the policy.

   Use DBMS_RLS.ADD_POLICY to define the policy.

   Use these specifications for the parameter values:

```
object_schema   OE
object_name     ORDERS
policy_name     OE_ORDERS_ACCESS_POLICY
function_schema OE
policy_function SALES_ORDERS_PKG.THE_PREDICATE
statement_types SELECT, INSERT, UPDATE, DELETE
update_check    FALSE,
enable          TRUE);
```

5. Connect as SYSDBA and create a logon trigger to implement fine-grained access control. You can call the trigger SET_ID_ON_LOGON. This trigger causes the context to be set as each user is logged on.

Practice 6 (continued)

6. Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         154         10
```

Note

During debugging, you may need to disable or remove some of the objects created for this lesson.

If you need to disable the logon trigger, issue the command:

```
ALTER TRIGGER set_id_on_logon DISABLE;
```

If you need to remove the policy you created, issue the command:

```
EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS', -
   'OE_ORDERS_ACCESS_POLICY')
```