C

# Studies for Implementing Triggers

## Objectives

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

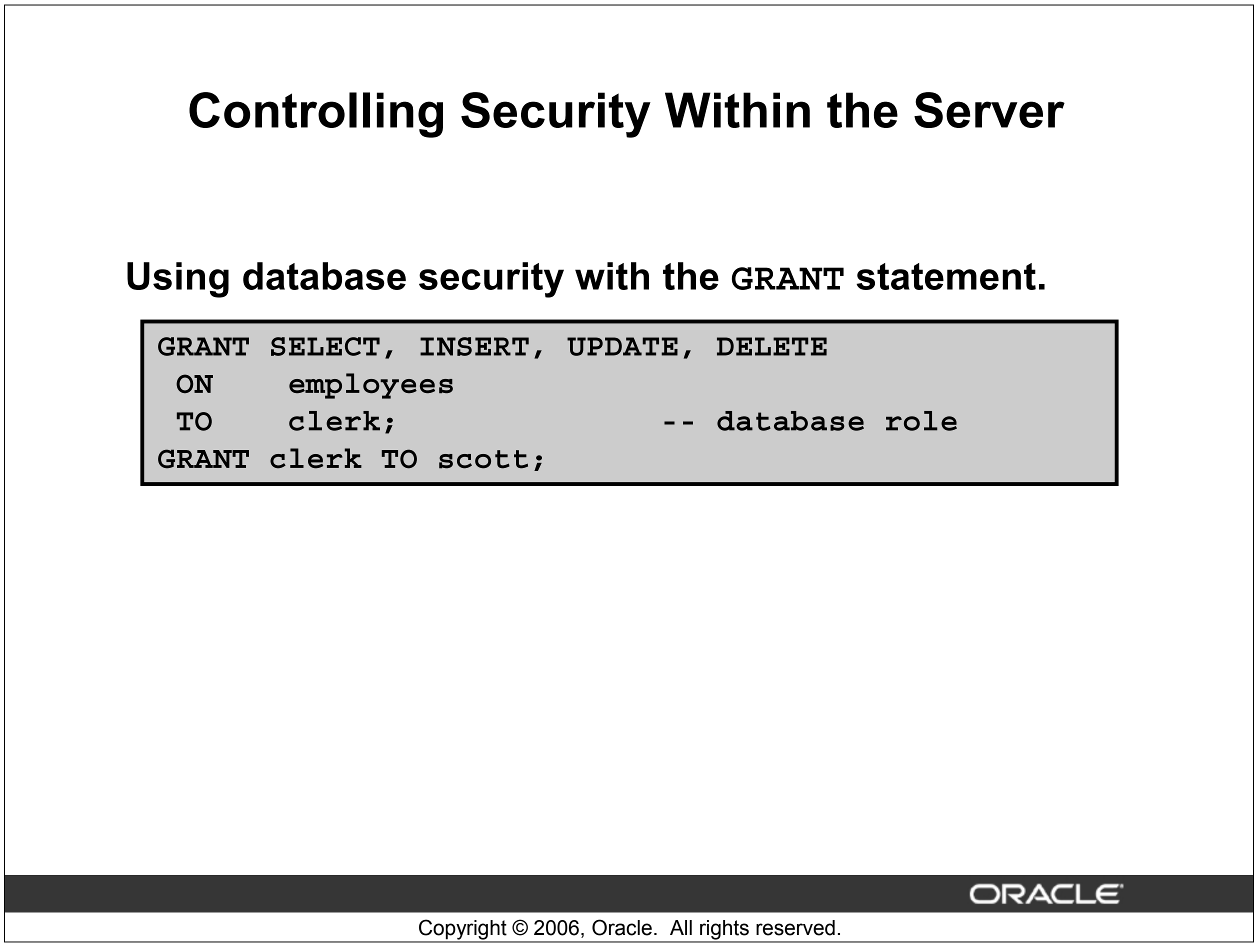
- **Enhance database security with triggers**
- **Audit data changes using DML triggers**
- **Enforce data integrity with DML triggers**
- **Maintain referential integrity using triggers**
- **Use triggers to replicate data between tables**
- **Use triggers to automate computation of derived data**
- **Provide event-logging capabilities using triggers**

### Lesson Aim

In this lesson, you learn to develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server. In some cases, it may be sufficient to refrain from using triggers and accept the functionality provided by the Oracle server.

This lesson covers the following business application scenarios:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging



## Controlling Security Within the Server

**Using database security with the `GRANT` statement.**

```
GRANT SELECT, INSERT, UPDATE, DELETE
 ON    employees
 TO    clerk;              -- database role
GRANT clerk TO scott;
```

### Controlling Security Within the Server

Develop schemas and roles within the Oracle server to control the security of data operations on tables according to the identity of the user.

- Base privileges upon the username supplied when the user connects to the database.
- Determine access to tables, views, synonyms, and sequences.
- Determine query, data-manipulation, and data-definition privileges.

## Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
dummy PLS_INTEGER;
BEGIN
 IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT','SUN')) THEN
   RAISE_APPLICATION_ERROR(-20506,'You may only
     change data during normal business hours.');
 END IF;
 SELECT COUNT(*) INTO dummy FROM holiday
 WHERE holiday_date = TRUNC (SYSDATE);
 IF dummy > 0 THEN
   RAISE_APPLICATION_ERROR(-20507,
     'You may not change data on a holiday.');
 END IF;
END;
/
```

ORACLE

### Controlling Security with a Database Trigger

Develop triggers to handle more complex security requirements.

- Base privileges on any database values, such as the time of day, the day of the week, and so on.
- Determine access to tables only.
- Determine data-manipulation privileges only.

## Using the Server Facility to Audit Data Operations

**The Oracle server stores the audit information in a data dictionary table or an operating system file.**

```
AUDIT INSERT, UPDATE, DELETE
  ON  departments
  BY ACCESS
WHENEVER SUCCESSFUL;
```

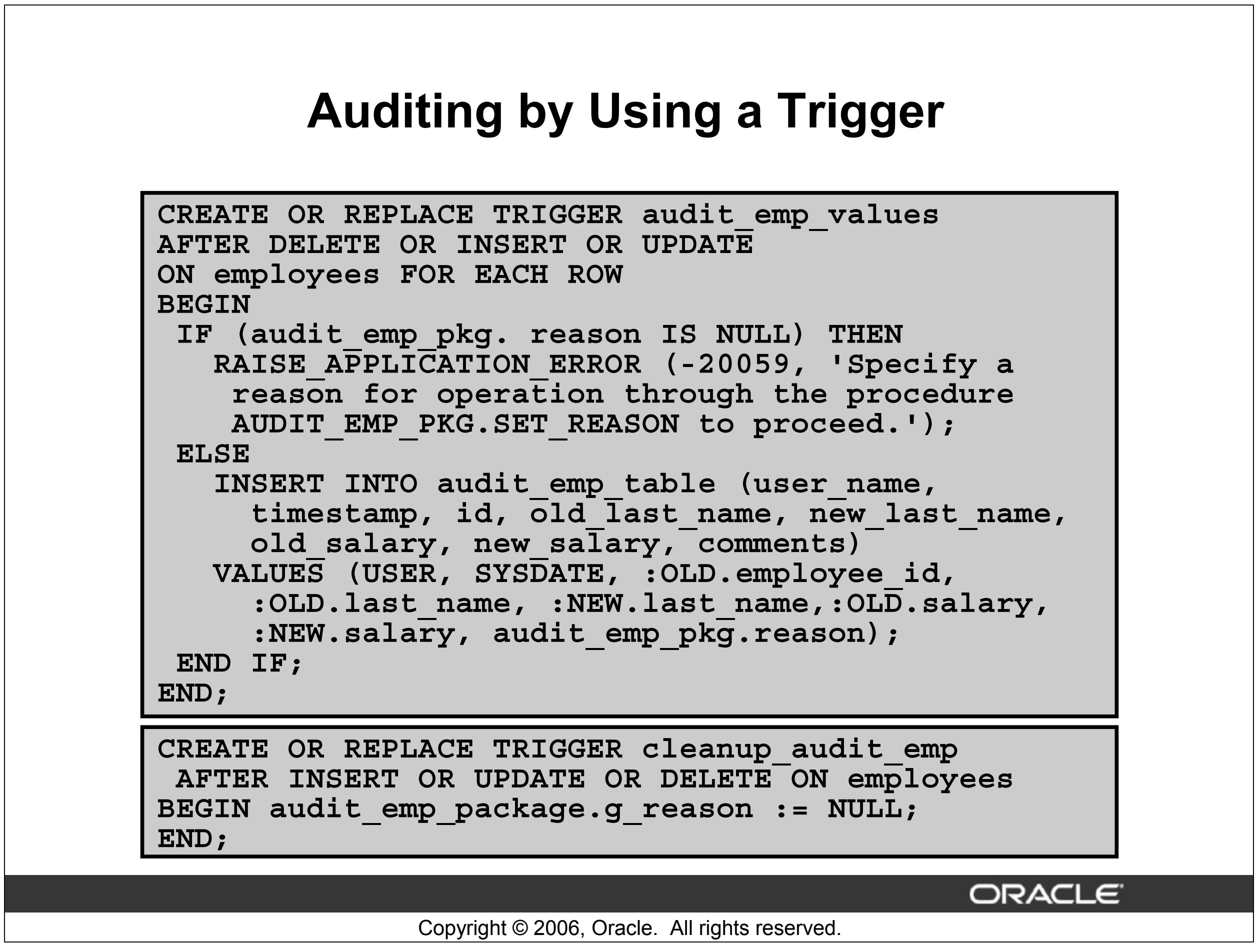
```
Audit succeeded.
```

### Auditing Data Operations

You can audit data operations within the Oracle server. Database auditing is used to monitor and gather data about specific database activities. The DBA can gather statistics such as which tables are being updated, how many I/Os are performed, how many concurrent users connect at peak time, and so on.

- Audit users, statements, or objects.
- Audit data retrieval, data-manipulation, and data-definition statements.
- Write the audit trail to a centralized audit table.
- Generate audit records once per session or once per access attempt.
- Capture successful attempts, unsuccessful attempts, or both.
- Enable and disable dynamically.

Executing SQL through PL/SQL program units may generate several audit records because the program units may refer to other database objects.



## Auditing by Using a Trigger

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE
ON employees FOR EACH ROW
BEGIN
 IF (audit_emp_pkg. reason IS NULL) THEN
   RAISE_APPLICATION_ERROR (-20059, 'Specify a
    reason for operation through the procedure
    AUDIT_EMP_PKG.SET_REASON to proceed.');
 ELSE
   INSERT INTO audit_emp_table (user_name,
     timestamp, id, old_last_name, new_last_name,
     old_salary, new_salary, comments)
   VALUES (USER, SYSDATE, :OLD.employee_id,
     :OLD.last_name, :NEW.last_name,:OLD.salary,
     :NEW.salary, audit_emp_pkg.reason);
 END IF;
END;
```

```
CREATE OR REPLACE TRIGGER cleanup_audit_emp
 AFTER INSERT OR UPDATE OR DELETE ON employees
BEGIN audit_emp_package.g_reason := NULL;
END;
```

### Auditing Data Values

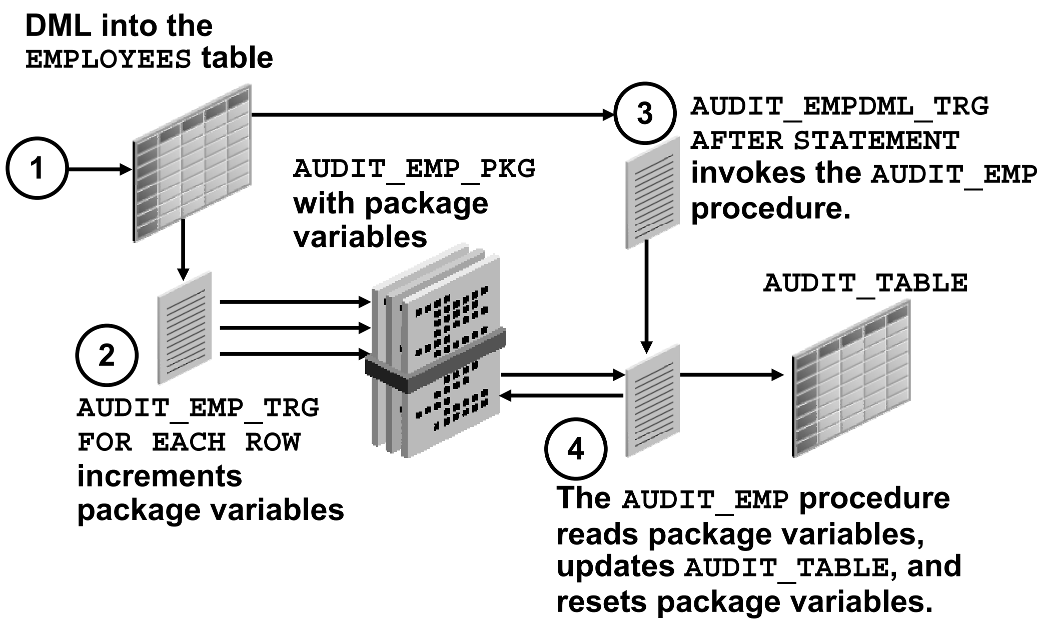
Audit actual data values with triggers.

You can do the following:

- Audit data-manipulation statements only.
- Write the audit trail to a user-defined audit table.
- Generate audit records once for the statement or once for each row.
- Capture successful attempts only.
- Enable and disable dynamically.

Using the Oracle server, you can perform database auditing. Database auditing cannot record changes to specific column values. If the changes to the table columns need to be tracked and column values need to be stored for each change, then use application auditing. Application auditing can be done either through stored procedures or database triggers, as shown in the example in the slide.

## Auditing Triggers by Using Package Constructs


DML into the EMPLOYEES table
1
AUDIT_EMP_PKG with package variables
3
AUDIT_EMPDML_TRG AFTER STATEMENT invokes the AUDIT_EMP procedure.
AUDIT_TABLE
2
AUDIT_EMP_TRG FOR EACH ROW increments package variables
4
The AUDIT_EMP procedure reads package variables, updates AUDIT_TABLE, and resets package variables.

### Auditing Triggers by Using Package Constructs

The following pages cover PL/SQL subprograms with examples of the interaction of triggers, package procedures, functions, and global variables.

The sequence of events:

1. Execute an `INSERT`, `UPDATE`, or `DELETE` command that can manipulate one or many rows.
2. `AUDIT_EMP_TRG` (the `AFTER ROW` trigger) calls the package procedure to increment the global variables in the `VAR_PACK` package. Because this is a row trigger, the trigger fires once for each row that you updated.
3. When the statement has finished, `AUDIT_EMP_TAB` (the `AFTER STATEMENT` trigger) calls the `AUDIT_EMP` procedure.
4. This procedure assigns the values of the global variables into local variables using the package functions, updates the `AUDIT_TABLE`, and then resets the global variables.

## Auditing Triggers by Using Package Constructs

**`AFTER` statement trigger:**

```
CREATE OR REPLACE TRIGGER audit_empdml_trg
AFTER UPDATE OR INSERT OR DELETE on employees
BEGIN
  audit_emp;      -- write the audit data
END audit_emp_tab;
/
```

**`AFTER` row trigger:**

```
CREATE OR REPLACE TRIGGER audit_emp_trg
AFTER UPDATE OR INSERT OR DELETE ON EMPLOYEES
FOR EACH ROW
-- Call Audit package to maintain counts
CALL audit_emp_pkg.set(INSERTING,UPDATING,DELETING);
/
```

### Auditing Triggers by Using Package Constructs (continued)

The AUDIT_EMP_TRIG trigger is a row trigger that fires after every row is manipulated. This trigger invokes the package procedures depending on the type of data manipulation language (DML) performed. For example, if the DML updates the salary of an employee, then the trigger invokes the SET_G_UP_SAL procedure. This package procedure, in turn, invokes the G_UP_SAL function. This function increments the GV_UP_SAL package variable that keeps account of the number of rows being changed due to the update of the salary.

The AUDIT_EMP_TAB trigger fires after the statement has finished. This trigger invokes the AUDIT_EMP procedure, which is explained on the following pages. The AUDIT_EMP procedure updates the AUDIT_TABLE table. An entry is made into the AUDIT_TABLE table with information such as the user who performed the DML, the table on which DML is performed, and the total number of such data manipulations performed so far on the table (indicated by the value of the corresponding column in the AUDIT_TABLE table). At the end, the AUDIT_EMP procedure resets the package variables to 0.

## AUDIT_PKG Package

```
CREATE OR REPLACE PACKAGE audit_emp_pkg IS
  delcnt PLS_INTEGER := 0;
  inscnt PLS_INTEGER := 0;
  updcnt  PLS_INTEGER := 0;
  PROCEDURE init;
  PROCEDURE set(i BOOLEAN,u BOOLEAN,d BOOLEAN);
END audit_emp_pkg;
/
CREATE OR REPLACE PACKAGE BODY audit_emp_pkg IS
 PROCEDURE init IS
  BEGIN
    inscnt := 0; updcnt := 0; delcnt := 0;
  END;
  PROCEDURE set(i BOOLEAN,u BOOLEAN,d BOOLEAN) IS
  BEGIN
    IF i THEN inscnt := inscnt + 1;
    ELSIF d THEN delcnt := delcnt + 1;
    ELSE upd := updcnt + 1;
    END IF;
  END;
END audit_emp_pkg;
/
```

### AUDIT_PKG Package

The AUDIT_PKG package declares public package variables (inscnt, updcnt, and delcnt) that are used to track the number of INSERT, UPDATE, and DELETE operations performed. In the code example, they are declared publicly for simplicity. However, it may be better to declare them as private variables to prevent them from being directly modified. If the variables are declared privately, in the package body, you would have to provide additional public subprograms to return their values to the user of the package.

The init procedure is used to initialize the public package variables to zero.

The set procedure accepts three BOOLEAN arguments: i, u, and d for an INSERT, UPDATE, or DELETE operation, respectively. The appropriate parameter value is set to TRUE when the trigger that invokes the set procedure is fired during one of the DML operations. A package variable is incremented by a value of 1, depending on which argument value is TRUE when the set procedure is invoked.

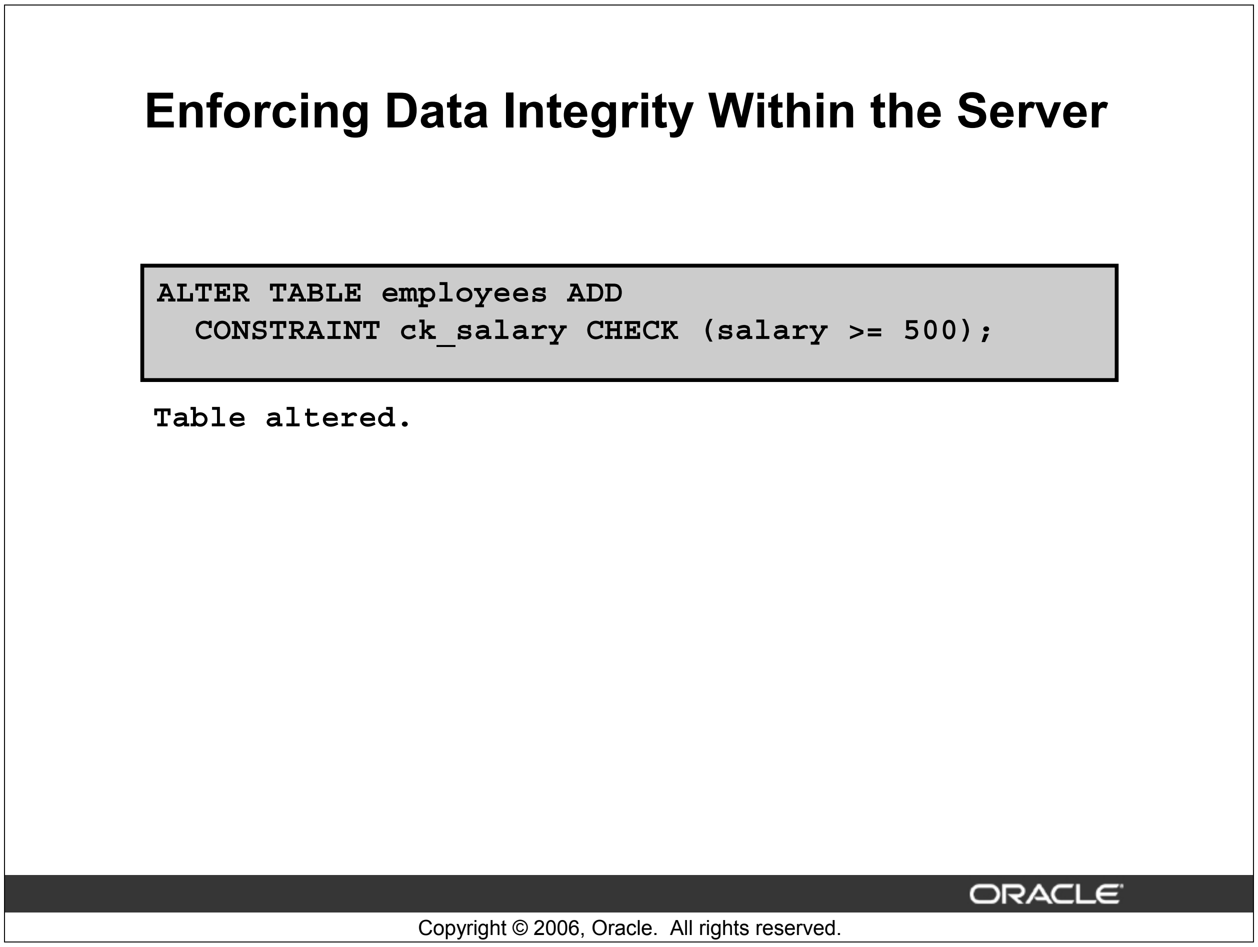
**Note:** A DML trigger can fire once for each DML on each row. Therefore, only one of the three variables passed to the set procedure can be TRUE at a given time. The remaining two arguments will be set to the value FALSE.

## AUDIT_TABLE Table and AUDIT_EMP Procedure

```
CREATE TABLE audit_table (
 USER_NAME   VARCHAR2(30),
 TABLE_NAME  VARCHAR2(30),
 INS         NUMBER,
 UPD         NUMBER,
 DEL         NUMBER)
/
CREATE OR REPLACE PROCEDURE audit_emp IS
BEGIN
  IF  delcnt + inscnt + updcnt <> 0 THEN
    UPDATE audit_table
     SET del = del + audit_emp_pkg.delcnt,
         ins = ins + audit_emp_pkg.inscnt,
         upd = upd + audit_emp_pkg.updcnt
    WHERE user_name = USER
    AND   table_name = 'EMPLOYEES';
    audit_emp_pkg.init;
  END IF;
END audit_emp;
/
```

### AUDIT_TABLE Table and AUDIT_EMP Procedure

The AUDIT_EMP procedure updates the AUDIT_TABLE table and calls the functions in the AUDIT_EMP_PKG package that reset the package variables, ready for the next DML statement.



## Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD
  CONSTRAINT ck_salary CHECK (salary >= 500);
```

```
Table altered.
```

### Enforcing Data Integrity Within the Server

You can enforce data integrity within the Oracle server and develop triggers to handle more complex data integrity rules.

The standard data integrity rules are not null, unique, primary key, and foreign key.

Use these rules to:

- Provide constant default values
- Enforce static constraints
- Enable and disable dynamically

**Example**

The code sample in the slide ensures that the salary is at least $500.

## Protecting Data Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20508,
      'Do not decrease salary.');
END;
/
```

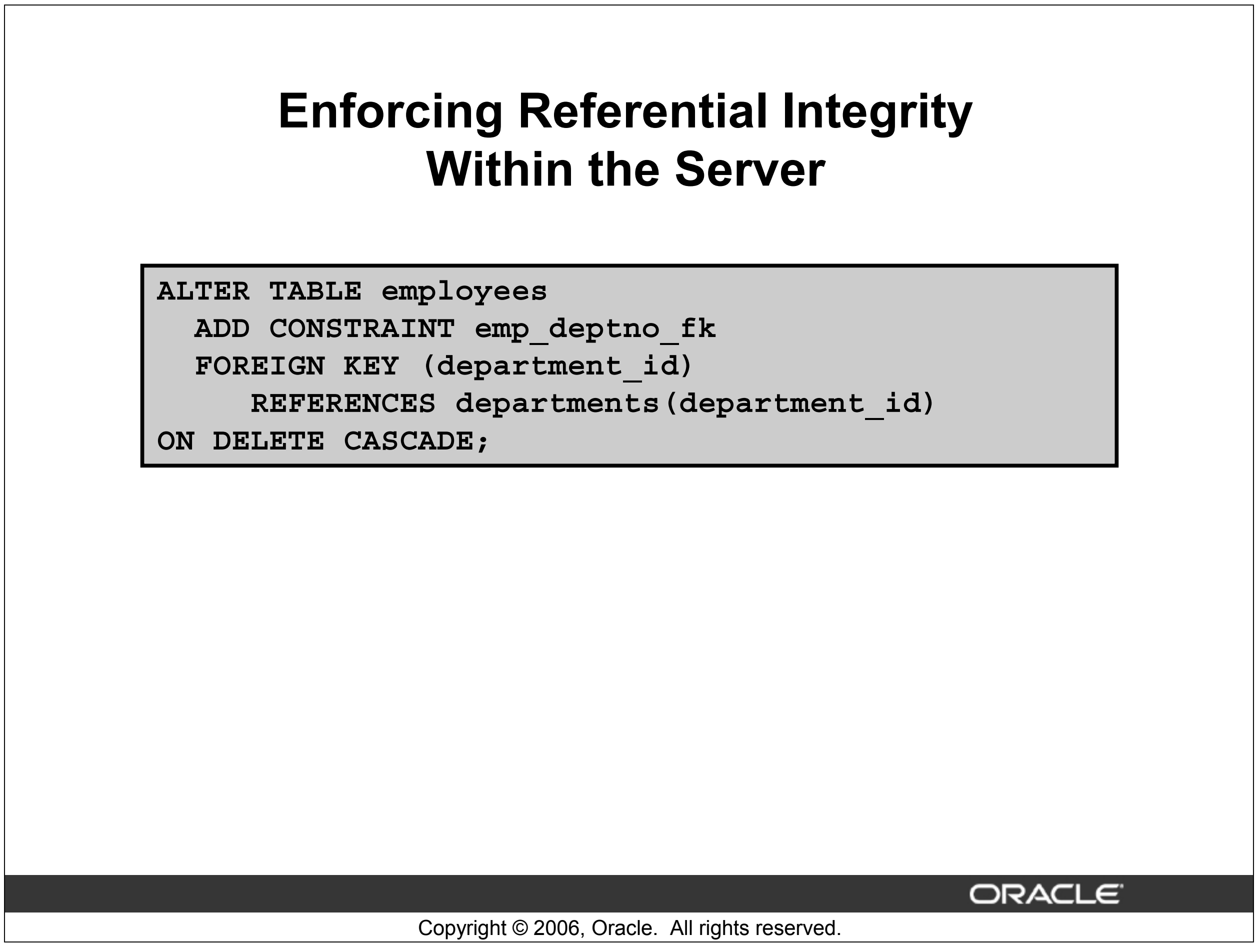
### Protecting Data Integrity with a Trigger

Protect data integrity with a trigger and enforce nonstandard data integrity checks.

- Provide variable default values.
- Enforce dynamic constraints.
- Enable and disable dynamically.
- Incorporate declarative constraints within the definition of a table to protect data integrity.

**Example**

The code sample in the slide ensures that the salary is never decreased.



## Enforcing Referential Integrity Within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
  FOREIGN KEY (department_id)
     REFERENCES departments(department_id)
ON DELETE CASCADE;
```

### Enforcing Referential Integrity Within the Server

Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency and enforce referential integrity within the server.

- Restrict updates and deletes.
- Cascade deletes.
- Enable and disable dynamically.

**Example**

When a department is removed from the `DEPARTMENTS` parent table, cascade the deletion to the corresponding rows in the `EMPLOYEES` child table.

## Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
 AFTER UPDATE OF department_id ON departments
 FOR EACH ROW
BEGIN
 UPDATE employees
  SET employees.department_id=:NEW.department_id
  WHERE employees.department_id=:OLD.department_id;
 UPDATE job_history
  SET department_id=:NEW.department_id
  WHERE department_id=:OLD.department_id;
END;
/
```

### Protecting Referential Integrity with a Trigger

The following referential integrity rules are not supported by declarative constraints:

- Cascade updates.
- Set to NULL for updates and deletions.
- Set to a default value on updates and deletions.
- Enforce referential integrity in a distributed system.
- Enable and disable dynamically.

You can develop triggers to implement these integrity rules.

**Example**

Enforce referential integrity with a trigger. When the value of DEPARTMENT_ID changes in the DEPARTMENTS parent table, cascade the update to the corresponding rows in the EMPLOYEES child table.

For a complete referential integrity solution using triggers, a single trigger is not enough.

## Replicating a Table Within the Server

```
CREATE MATERIALIZED VIEW emp_copy
  NEXT sysdate + 7
  AS SELECT * FROM employees@ny;
```

### Creating a Materialized View

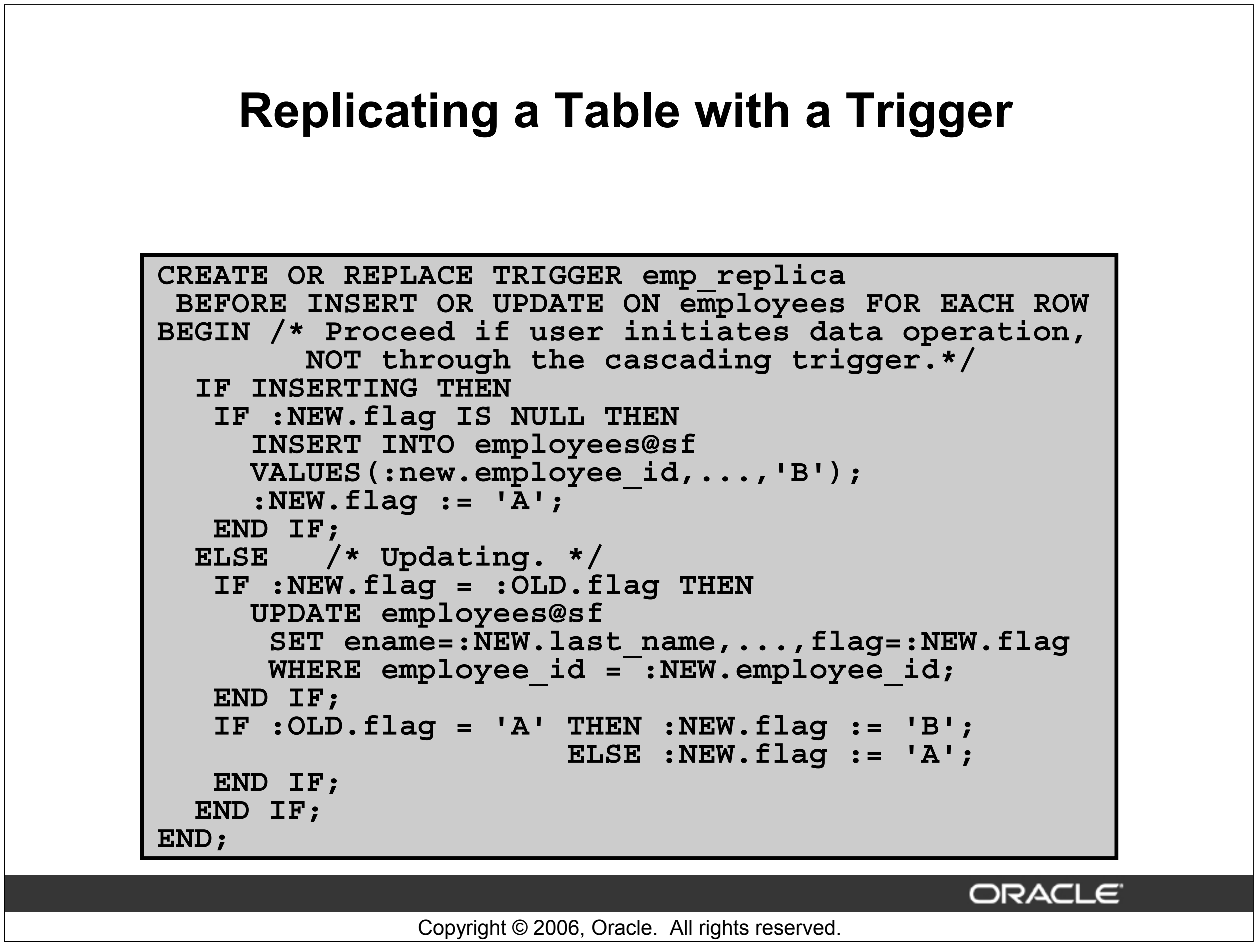
Materialized views enable you to maintain copies of remote data on your local node for replication purposes. You can select data from a materialized view as you would from a normal database table or view. A materialized view is a database object that contains the results of a query, or a copy of some database on a query. The `FROM` clause of the query of a materialized view can name tables, views, and other materialized views.

When a materialized view is used, replication is performed implicitly by the Oracle server. This performs better than using user-defined PL/SQL triggers for replication. Materialized views:

- Copy data from local and remote tables asynchronously, at user-defined intervals
- Can be based on multiple master tables
- Are read-only by default, unless using the Oracle Advanced Replication feature
- Improve the performance of data manipulation on the master table

Alternatively, you can replicate tables using triggers.

The example in the slide creates a copy of the remote `EMPLOYEES` table from New York. The `NEXT` clause specifies a date time expression for the interval between automatic refreshes.



## Replicating a Table with a Trigger

```
CREATE OR REPLACE TRIGGER emp_replica
 BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN /* Proceed if user initiates data operation,
        NOT through the cascading trigger.*/
  IF INSERTING THEN
   IF :NEW.flag IS NULL THEN
     INSERT INTO employees@sf
     VALUES(:new.employee_id,...,'B');
     :NEW.flag := 'A';
   END IF;
  ELSE   /* Updating. */
   IF :NEW.flag = :OLD.flag THEN
     UPDATE employees@sf
      SET ename=:NEW.last_name,...,flag=:NEW.flag
      WHERE employee_id = :NEW.employee_id;
   END IF;
   IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
                      ELSE :NEW.flag := 'A';
   END IF;
  END IF;
END;
```

### Replicating a Table with a Trigger

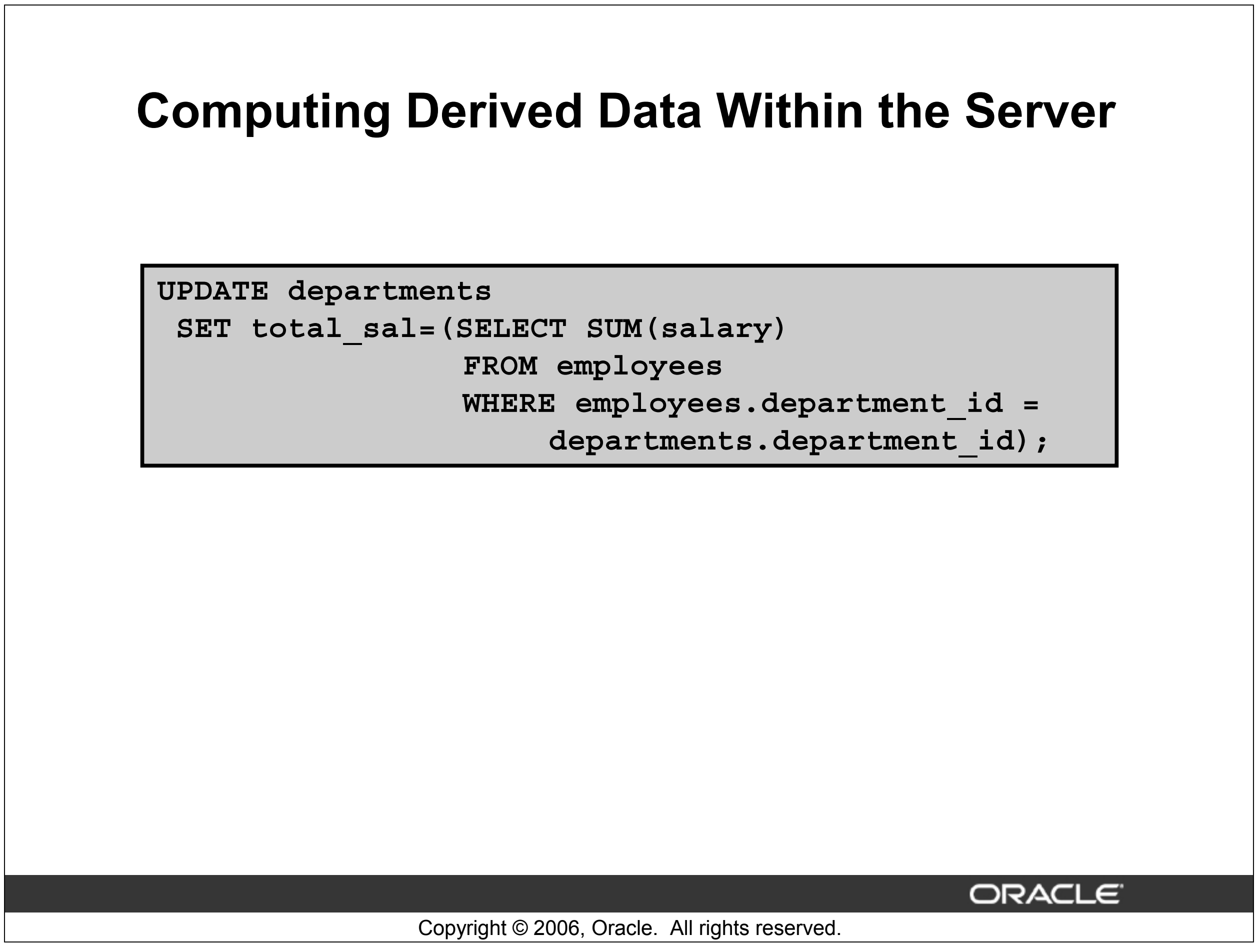
You can replicate a table with a trigger. By replicating a table, you can:

- Copy tables synchronously, in real time
- Base replicas on a single master table
- Read from replicas as well as write to them

**Note:** Excessive use of triggers can impair the performance of data manipulation on the master table, particularly if the network fails.

**Example**

In New York, replicate the local `EMPLOYEES` table to San Francisco.



## Computing Derived Data Within the Server

```
UPDATE departments
 SET total_sal=(SELECT SUM(salary)
                FROM employees
                WHERE employees.department_id =
                    departments.department_id);
```

### Computing Derived Data Within the Server

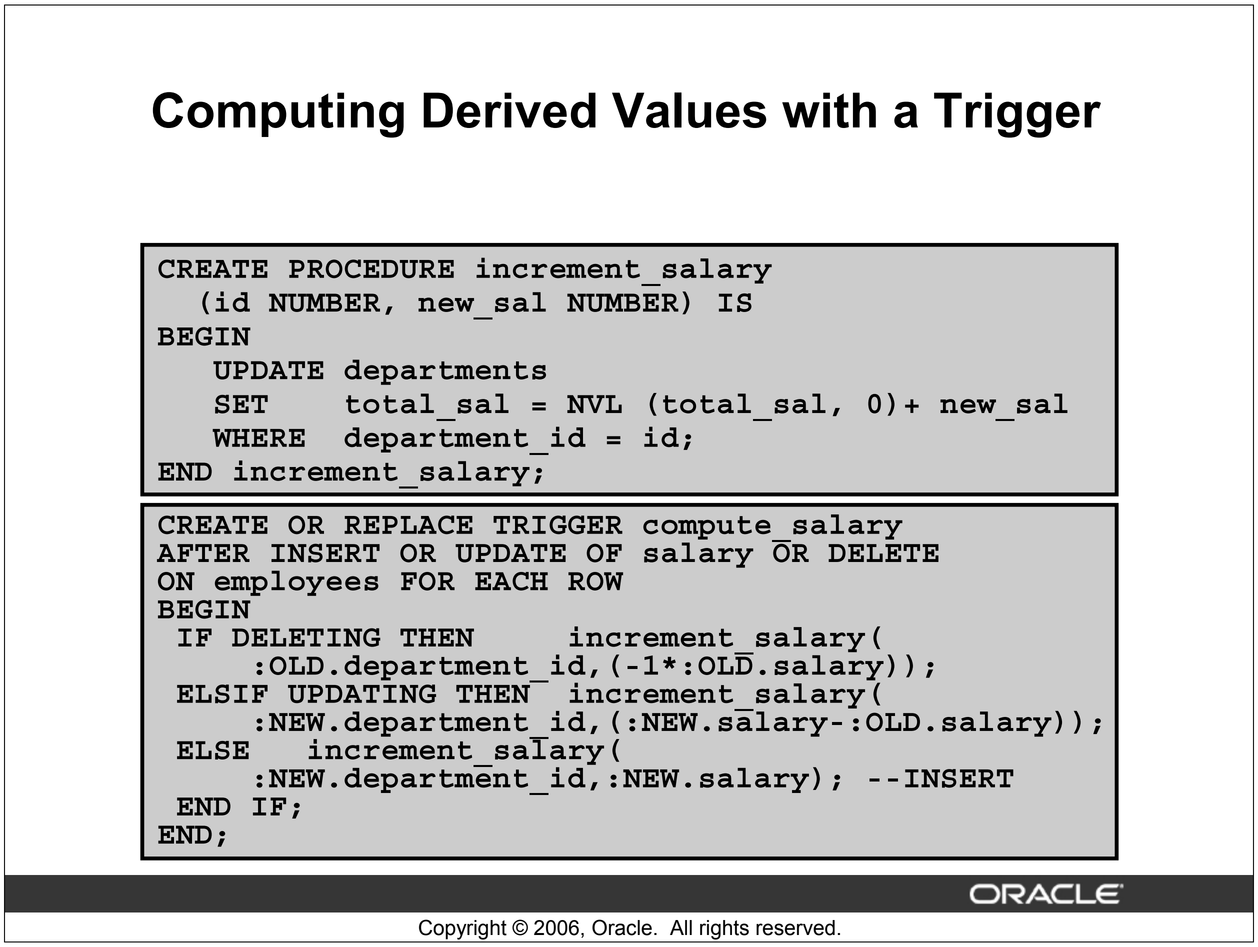
By using the server, you can schedule batch jobs or use the database Scheduler for the following scenarios:

- Compute derived column values asynchronously, at user-defined intervals.
- Store derived values only within database tables.
- Modify data in one pass to the database and calculate derived data in a second pass.

Alternatively, you can use triggers to keep running computations of derived data.

**Example**

Keep the salary total for each department within a special `TOTAL_SALARY` column of the `DEPARTMENTS` table.



## Computing Derived Values with a Trigger

```
CREATE PROCEDURE increment_salary
  (id NUMBER, new_sal NUMBER) IS
BEGIN
   UPDATE departments
   SET    total_sal = NVL (total_sal, 0)+ new_sal
   WHERE  department_id = id;
END increment_salary;
```

```
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE
ON employees FOR EACH ROW
BEGIN
 IF DELETING THEN     increment_salary(
     :OLD.department_id,(-1*:OLD.salary));
 ELSIF UPDATING THEN  increment_salary(
     :NEW.department_id,(:NEW.salary-:OLD.salary));
 ELSE   increment_salary(
     :NEW.department_id,:NEW.salary); --INSERT
 END IF;
END;
```

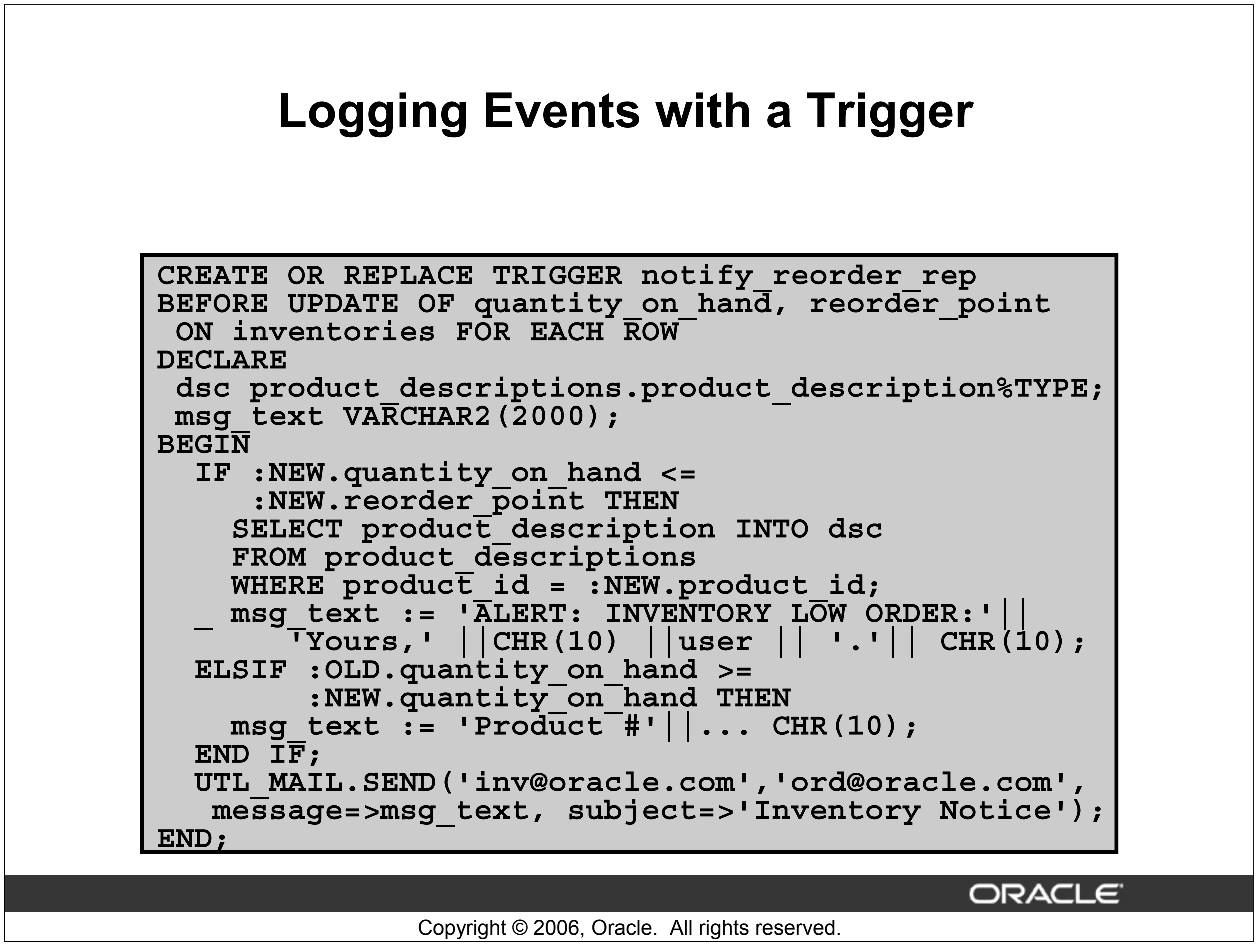
### Computing Derived Data Values with a Trigger

By using a trigger, you can perform the following tasks:

- Compute derived columns synchronously, in real time.
- Store derived values within database tables or within package global variables.
- Modify data and calculate derived data in a single pass to the database.

**Example**

Keep a running total of the salary for each department in the special `TOTAL_SALARY` column of the `DEPARTMENTS` table.



## Logging Events with a Trigger

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
 ON inventories FOR EACH ROW
DECLARE
 dsc product_descriptions.product_description%TYPE;
 msg_text VARCHAR2(2000);
BEGIN
  IF :NEW.quantity_on_hand <=
     :NEW.reorder_point THEN
    SELECT product_description INTO dsc
    FROM product_descriptions
    WHERE product_id = :NEW.product_id;
  _ msg_text := 'ALERT: INVENTORY LOW ORDER:'||
       'Yours,' ||CHR(10) ||user || '.'|| CHR(10);
  ELSIF :OLD.quantity_on_hand >=
        :NEW.quantity_on_hand THEN
    msg_text := 'Product #'||... CHR(10);
  END IF;
  UTL_MAIL.SEND('inv@oracle.com','ord@oracle.com',
   message=>msg_text, subject=>'Inventory Notice');
END;
```

ORACLE

### Logging Events with a Trigger

In the server, you can log events by querying data and performing operations manually. This sends an e-mail message when the inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package `UTL_MAIL` to send the e-mail message.

**Logging Events Within the Server**

1. Query data explicitly to determine whether an operation is necessary.
2. Perform the operation, such as sending a message.

**Using Triggers to Log Events**

1. Perform operations implicitly, such as firing off an automatic electronic memo.
2. Modify data and perform its dependent operation in a single step.
3. Log events automatically as data is changing.

## Logging Events with a Trigger (continued)

### Logging Events Transparently

In the trigger code:

- `CHR(10)` is a carriage return
- `Reorder_point` is not NULL
- Another transaction can receive and read the message in the pipe

### Example

```
 CREATE OR REPLACE TRIGGER notify_reorder_rep
 BEFORE UPDATE OF amount_in_stock, reorder_point
 ON inventory  FOR EACH ROW
 DECLARE
   dsc product.descrip%TYPE;
   msg_text VARCHAR2(2000);
 BEGIN
   IF  :NEW.amount_in_stock <= :NEW.reorder_point THEN
     SELECT descrip INTO  dsc
     FROM PRODUCT WHERE prodid = :NEW.product_id;
     msg_text := 'ALERT: INVENTORY LOW ORDER:'||CHR(10)||
     'It has come to my personal attention that, due to recent'
     ||CHR(10)||'transactions, our inventory for product # '||
     TO_CHAR(:NEW.product_id)||'-- '|| dsc ||
     ' -- has fallen below acceptable levels.' || CHR(10) ||
     'Yours,' ||CHR(10) ||user || '.'|| CHR(10)|| CHR(10);
   ELSIF :OLD.amount_in_stock >= :NEW.amount_in_stock THEN
     msg_text := 'Product #'|| TO_CHAR(:NEW.product_id)
     ||' ordered. '|| CHR(10)|| CHR(10);
   END IF;
   UTL_MAIL.SEND('inv@oracle.com', 'ord@oracle.com',
      message => msg_text, subject => 'Inventory Notice');
END;
```

## Summary

In this lesson, you should have learned how to:

- Use database triggers and database server functionality to:
  - Enhance database security
  - Audit data changes
  - Enforce data integrity
  - Maintain referential integrity
  - Replicate data between tables
  - Automate computation of derived data
  - Provide event-logging capabilities
- Recognize when to use triggers to database functionality

### Summary

This lesson provides some detailed comparison of using the Oracle database server functionality to implement security, auditing, data integrity, replication, and logging. The lesson also covers how database triggers can be used to implement the same features but go further to enhance the features that the database server provides. In some cases, you must use a trigger to perform some activities (such as computation of derived data) because the Oracle server cannot know how to implement this kind of business rule without some programming effort.