9

Developing a Simple Oracle Database Application

By following the instructions for developing this simple application, you learn the general procedure for developing Oracle Database applications.

About the Application

The application has the following purpose, structure, and naming conventions.

Purpose of the Application

The application is intended for two kinds of users in a company.

- Typical users (managers of employees)
- Application administrators

Typical users can do the following:

- Get the employees in a given department
- Get the job history for a given employee
- Show general information for a given employee (name, department, job, manager, salary, and so on)
- Change the salary of a given employee
- Change the job of a given employee

Application administrators can do the following:

- Change the ID, title, or salary range of an existing job
- Add a new job
- Change the ID, name, or manager of an existing department
- Add a new department

Structure of the Application

The application uses the following schema objects and schemas.

Schema Objects of the Application

The application is composed of these schema objects:

- Four tables, which store data about:
  - Jobs
  - Departments

  - Employees
  - Job history of employees
- Four editioning views, which cover the tables, enabling you to use edition-based redefinition (EBR) to upgrade the finished application when it is in use
- Two triggers, which enforce business rules
- Two sequences that generate unique primary keys for new departments and new employees
- Two packages:
  - employees_pkg, the application program interface (API) for typical users
  - admin_pkg, the API for application administrators

  The typical users and application administrators access the application only through its APIs. Therefore, they can change the data only by invoking package subprograms.

> **See Also:**
>
> - "About Oracle Database" for information about schema objects
> - *Oracle Database Development Guide* for information about EBR

Schemas for the Application

For security, the application uses these five schemas (or users), each of which has *only* the privileges that it needs:

- The `app_data` schema, which owns all of the schema objects except the packages and loads its tables with data from tables in the `hr` sample schema.

  The developers who create the packages never work in this schema. Therefore, they cannot accidentally alter or drop application schema objects.
- The `app_code` schema, which owns only the package `employees_pkg`.

  The developers of the `employees_pkg` package work in this schema.
- The `app_admin` schema, which owns only the package `admin_pkg`.

  The developers of the `admin_pkg` package work in this schema.
- The `app_user` user, the typical application user, who owns nothing and can only run the package `employees_pkg`.

  The middle-tier application server connects to the database in the connection pool as user `app_user`. If this schema is compromised—by a SQL injection bug, for example—the attacker can see and change only what the `employees_pkg` package subprograms let it see and change. The attacker cannot drop tables, escalate privileges, create or alter schema objects, or anything else.
- The `app_admin_user` user, an application administrator, who owns nothing and can only run the `admin_pkg` and `employees_pkg` packages.

The connection pool for this schema is very small, and only privileged users can access it. If this schema is compromised, the attacker can see and change only what `admin_pkg` and `employees_pkg` package subprograms let it see and change.

Suppose that instead of users `app_user` and `app_admin_user`, the application had only one schema that owned nothing and could run both `employees_pkg` and `admin_pkg` packages. The connection pool for this schema would have to be large enough for both the typical users and the application administrators. If there were a SQL injection bug in the `employees_pkg` package, a typical user who exploited that bug could access the `admin_pkg` package.

Suppose that instead of the `app_data`, `app_code`, and `app_admin` schemas, the application had only one schema that owned all the schema objects, including the packages. The packages would then have all privileges on the tables, which would be both unnecessary and undesirable.

For example, suppose that you have an audit trail table, `AUDIT_TRAIL`. You want the developers of the `employees_pkg` package to be able to write to the `AUDIT_TRAIL` table, but not read or change it. You want the developers of the `admin_pkg` package to be able to read the `AUDIT_TRAIL` table and write to it, but not change it. If the `AUDIT_TRAIL` table and the `employees_pkg`, and `admin_pkg` packages belong to the same schema, then the developers of the two packages have all privileges on the `AUDIT_TRAIL` table. However, if the `AUDIT_TRAIL` table belongs to the `app_data` schema, the `employees_pkg` package belongs to the `app_code` schema, and the `admin_pkg` package belongs to the `app_admin` schema, then you can connect to the database as the `app_data` schema and run the following commands:

```
GRANT INSERT ON AUDIT_TRAIL TO app_code;
GRANT INSERT, SELECT ON AUDIT_TRAIL TO app_admin;
```

See Also:

- "About Oracle Database" for information about schemas
- "About Sample Schema HR" for information about sample schema `HR`
- "Recommended Security Practices"

Naming Conventions in the Application

The application uses these naming conventions.

Item	Name
Table	*table#*
Editioning view for *table#*	*table*
Trigger on editioning view *table*	*table*_{a|b}*event*[_fer] where: • a identifies an AFTER trigger. • b identifies a BEFORE trigger. • fer identifies a FOR EACH ROW trigger. • *event* identifies the event that fires the trigger. For example: i for INSERT, iu for INSERT or UPDATE, d for DELETE.
PRIMARY KEY constraint in *table#*	*table*_pk

Item	Name
NOT NULL constraint on *table#.column*	*table_column*_not_null[1]
UNIQUE constraint on *table#.column*	*table_column*_unique[1]
CHECK constraint on *table#.column*	*table_column*_check[1]
REF constraint on *table1#.column* to *table2#.column*	*table1*_to_*table2*_fk[1]
REF constraint on *table1#.column1* to *table2#.column2*	*table1_col1*_to_*table2_col2*_fk[1] [2]
Sequence for *table#*	*table_sequence*
Parameter name	p_*name*
Local variable name	l_*name*

1 *table*, *table1*, and *table2* are abbreviated to emp for employees, dept for departments, and job_hist for job_history.
2 *col1* and *col2* are abbreviations of column names *column1* and *column2*. A constraint name cannot have more than 30 characters.

Creating the Schemas for the Application

Using the procedure in this section, create the schemas for the application.

The schema names are:

- app_data
- app_code
- app_admin
- app_user
- app_admin_user

> **Note:**
>
> For the following procedure, you need the name and password of a user who has the CREATE USER and DROP USER system privileges.

To create the schema (or user) *schema_name*:

1. Using SQL*Plus, connect to Oracle Database as a user with the CREATE USER and DROP USER system privileges.

   The `SQL>` prompt appears.

2. In case the schema exists, drop the schema and its objects with this SQL statement:

   ```
   DROP USER schema_name CASCADE;
   ```

   If the schema existed, the system responds:

   ```
   User dropped.
   ```

If the schema did not exist, the system responds:

```
DROP USER schema_name CASCADE
          *
ERROR at line 1:
ORA-01918: user 'schema_name' does not exist
```

3. If *schema_name* is either `app_data`, `app_code`, or `app_admin`, then create the schema with this SQL statement:

```
CREATE USER schema_name IDENTIFIED BY password
DEFAULT TABLESPACE USERS
QUOTA UNLIMITED ON USERS
ENABLE EDITIONS;
```

Otherwise, create the schema with this SQL statement:

```
CREATE USER schema_name IDENTIFIED BY password
ENABLE EDITIONS;
```

> **Caution:**
>
> Choose a secure password. For guidelines for secure passwords, see *Oracle Database Security Guide*.

The system responds:

```
User created.
```

4. (Optional) In SQL Developer, create a connection for the schema, using the instructions in "Connecting to Oracle Database from SQL Developer".

> **See Also:**
>
> - "About the Application"
> - "Connecting to Oracle Database from SQL*Plus"
> - *Oracle Database SQL Language Reference* for information about the `DROP USER` statement
> - *Oracle Database SQL Language Reference* for information about the `CREATE USER` statement

Granting Privileges to the Schemas

To grant privileges to schemas, use the SQL statement GRANT.

You can enter the GRANT statements either in SQL*Plus or in the Worksheet of SQL Developer. For security, grant each schema *only* the privileges that it needs.

> **See Also:**
>
> - "About the Application"
> - *Oracle Database SQL Language Reference* for information about the GRANT statement

Granting Privileges to the app_data Schema

Grant to the app_data schema *only* the privileges to do the following:

- Connect to Oracle Database:

```
GRANT CREATE SESSION TO app_data;
```

- Create the tables, views, triggers, and sequences for the application:

```
GRANT CREATE TABLE, CREATE VIEW, CREATE TRIGGER, CREATE SEQUENCE TO app_data;
```

- Load data from four tables in the sample schema HR into its own tables:

```
GRANT SELECT ON HR.DEPARTMENTS TO app_data;
GRANT SELECT ON HR.EMPLOYEES TO app_data;
GRANT SELECT ON HR.JOB_HISTORY TO app_data;
GRANT SELECT ON HR.JOBS TO app_data;
```

Granting Privileges to the app_code Schema

Grant to the app_code schema *only* the privileges to do the following:

- Connect to Oracle Database:

```
GRANT CREATE SESSION TO app_code;
```

- Create the package employees_pkg:

```
GRANT CREATE PROCEDURE TO app_code;
```

- Create a synonym (for convenience):

```
GRANT CREATE SYNONYM TO app_code;
```

Granting Privileges to the app_admin Schema

Grant to the app_admin schema *only* the privileges to do the following:

- Connect to Oracle Database:

```
GRANT CREATE SESSION TO app_admin;
```

- Create the package admin_pkg:

```
GRANT CREATE PROCEDURE TO app_admin;
```

- Create a synonym (for convenience):

```
GRANT CREATE SYNONYM TO app_admin;
```

Granting Privileges to the app_user and app_admin_user Schemas

Grant to the app_user and app_admin_user schemas *only* the privileges to do the following:

- Connect to Oracle Database:

```
GRANT CREATE SESSION TO app_user;
GRANT CREATE SESSION TO app_admin_user;
```

- Create synonyms (for convenience):

```
GRANT CREATE SYNONYM TO app_user;
GRANT CREATE SYNONYM TO app_admin_user;
```

Creating the Schema Objects and Loading the Data

This section shows how to create the tables, editioning views, triggers, and sequences for the application, how to load data into the tables, and how to grant privileges on these schema objects to the users that need them.

To create the schema objects and load the data:

1. Connect to Oracle Database as user app_data.

   For instructions, see either "Connecting to Oracle Database from SQL*Plus" or "Connecting to Oracle Database from SQL Developer".
2. Create the tables, with all necessary constraints except the foreign key constraint that you must add after you load the data.
3. Create the editioning views.
4. Create the triggers.
5. Create the sequences.
6. Load the data into the tables.
7. Add the foreign key constraint.

Creating the Tables

This section shows how to create the tables for the application, with all necessary constraints except one, which you must add after you load the data.

> **Note:**
>
> You must be connected to Oracle Database as user `app_data`.

In the following procedure, you can enter the statements either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the tables with the SQL Developer tool Create Table.

To create the tables:

1. Create jobs#, which stores information about the jobs in the company (one row for each job):

```
CREATE TABLE jobs#
( job_id      VARCHAR2(10)
              CONSTRAINT jobs_pk PRIMARY KEY,
  job_title   VARCHAR2(35)
              CONSTRAINT jobs_job_title_not_null NOT NULL,
  min_salary  NUMBER(6)
              CONSTRAINT jobs_min_salary_not_null NOT NULL,
  max_salary  NUMBER(6)
              CONSTRAINT jobs_max_salary_not_null NOT NULL
)
/
```

2. Create departments#, which stores information about the departments in the company (one row for each department):

```
CREATE TABLE departments#
( department_id    NUMBER(4)
                   CONSTRAINT departments_pk PRIMARY KEY,
  department_name  VARCHAR2(30)
                   CONSTRAINT department_name_not_null NOT NULL
                   CONSTRAINT department_name_unique UNIQUE,
  manager_id       NUMBER(6)
)
/
```

3. Create employees#, which stores information about the employees in the company (one row for each employee):

```
CREATE TABLE employees#
( employee_id     NUMBER(6)
                  CONSTRAINT employees_pk PRIMARY KEY,
  first_name      VARCHAR2(20)
                  CONSTRAINT emp_first_name_not_null NOT NULL,
  last_name       VARCHAR2(25)
                  CONSTRAINT emp_last_name_not_null NOT NULL,
  email_addr      VARCHAR2(25)
                  CONSTRAINT emp_email_addr_not_null NOT NULL,
  hire_date       DATE
                  DEFAULT TRUNC(SYSDATE)
                  CONSTRAINT emp_hire_date_not_null NOT NULL
                  CONSTRAINT emp_hire_date_check
                    CHECK(TRUNC(hire_date) = hire_date),
  country_code    VARCHAR2(5)
                  CONSTRAINT emp_country_code_not_null NOT NULL,
  phone_number    VARCHAR2(20)
                  CONSTRAINT emp_phone_number_not_null NOT NULL,
  job_id          CONSTRAINT emp_job_id_not_null NOT NULL
                  CONSTRAINT emp_jobs_fk REFERENCES jobs#,
  job_start_date  DATE
                  CONSTRAINT emp_job_start_date_not_null NOT NULL,
                  CONSTRAINT emp_job_start_date_check
                    CHECK(TRUNC(JOB_START_DATE) = job_start_date),
  salary          NUMBER(6)
                  CONSTRAINT emp_salary_not_null NOT NULL,
  manager_id      CONSTRAINT emp_mgr_to_empno_fk REFERENCES employees#,
  department_id   CONSTRAINT emp_to_dept_fk REFERENCES departments#
```

```
)
/
```

The reasons for the REF constraints are:

- An employee must have an existing job. That is, values in the column employees#.job_id must also be values in the column jobs#.job_id.
- An employee must have a manager who is also an employee. That is, values in the column employees#.manager_id must also be values in the column employees#.employee_id.
- An employee must work in an existing department. That is, values in the column employees#.department_id must also be values in the column departments#.department_id.

Also, the manager of an employee must be the manager of the department in which the employee works. That is, values in the column employees#.manager_id must also be values in the column departments#.manager_id. However, you could not specify the necessary constraint when you created departments#, because employees# did not exist yet. Therefore, you must add a foreign key constraint to departments# later (see "Adding the Foreign Key Constraint").

4. Create job_history#, which stores the job history of each employee in the company (one row for each job held by the employee):

```
CREATE TABLE job_history#
( employee_id  CONSTRAINT job_hist_to_employees_fk REFERENCES employees#,
  job_id       CONSTRAINT job_hist_to_jobs_fk REFERENCES jobs#,
  start_date   DATE
               CONSTRAINT job_hist_start_date_not_null NOT NULL,
  end_date     DATE
               CONSTRAINT job_hist_end_date_not_null NOT NULL,
  department_id
              CONSTRAINT job_hist_to_departments_fk REFERENCES departments#
              CONSTRAINT job_hist_dept_id_not_null NOT NULL,
              CONSTRAINT job_history_pk PRIMARY KEY(employee_id,start_date),
              CONSTRAINT job_history_date_check CHECK( start_date < end_date )
)
/
```

The reasons for the REF constraints are that the employee, job, and department must exist. That is:

- Values in the column job_history#.employee_id must also be values in the column employees#.employee_id.
- Values in the column job_history#.job_id must also be values in the column jobs#.job_id.
- Values in the column job_history#.department_id must also be values in the column departments#.department_id.

> **See Also:**
>
> "Creating Tables"

Creating the Editioning Views

> **Note:**
>
> You must be connected to Oracle Database as user app_data.

To create the editioning views, use the following statements (in any order). You can enter the statements either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the editioning views with the SQL Developer tool Create View.

```
CREATE OR REPLACE EDITIONING VIEW jobs AS SELECT * FROM jobs#
/
CREATE OR REPLACE EDITIONING VIEW departments AS SELECT * FROM departments#
/
CREATE OR REPLACE EDITIONING VIEW employees AS SELECT * FROM employees#
/
CREATE OR REPLACE EDITIONING VIEW job_history AS SELECT * FROM job_history#
/
```

> **Note:**
>
> The application must always reference the base tables through the editioning views. Otherwise, the editioning views do not cover the tables and you cannot use EBR to upgrade the finished application when it is in use.

> **See Also:**
>
> - "Creating Views"
> - *Oracle Database Development Guide* for general information about editioning views
> - *Oracle Database Development Guide* for information about preparing an application to use editioning views

Creating the Triggers

> **Note:**
>
> You must be connected to Oracle Database as user app_data.

The triggers in the application enforce these business rules:

- An employee with job *j* must have a salary between the minimum and maximum salaries for job *j*.
- If an employee with job *j* has salary *s*, then you cannot change the minimum salary for *j* to a value greater than *s* or the maximum salary for *j* to a value less than *s*. (To do so would make existing data invalid.)

> **See Also:**
>
> Using Triggers, for information about triggers

Creating the Trigger to Enforce the First Business Rule

The first business rule is: An employee with job *j* must have a salary between the minimum and maximum salaries for job *j*.

This rule could be violated either when a new row is inserted into the employees table or when the salary or job_id column of the employees table is updated.

To enforce the rule, create the following trigger on the editioning view employees. You can enter the CREATE TRIGGER statement either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the trigger with the SQL Developer tool Create Trigger.

```
CREATE OR REPLACE TRIGGER employees_aiufer
AFTER INSERT OR UPDATE OF salary, job_id ON employees FOR EACH ROW
DECLARE
  l_cnt NUMBER;
BEGIN
  LOCK TABLE jobs IN SHARE MODE;  -- Ensure that jobs does not change
                                 -- during the following query.
  SELECT COUNT(*) INTO l_cnt
  FROM jobs
  WHERE job_id = :NEW.job_id
  AND :NEW.salary BETWEEN min_salary AND max_salary;

  IF (l_cnt<>1) THEN
    RAISE_APPLICATION_ERROR( -20002,
      CASE
        WHEN :new.job_id = :old.job_id
        THEN 'Salary modification invalid'
        ELSE 'Job reassignment puts salary out of range'
      END );
  END IF;
END;
/
```

`LOCK TABLE jobs IN SHARE MODE` prevents other users from changing the table jobs while the trigger is querying it. Preventing changes to jobs during the query is necessary because nonblocking reads prevent the trigger from "seeing" changes that other users make to jobs while the trigger is changing employees (and prevent those users from "seeing" the changes that the trigger makes to employees).

Another way to prevent changes to jobs during the query is to include the FOR UPDATE clause in the SELECT statement. However, SELECT FOR UPDATE restricts concurrency more than LOCK TABLE jobs IN SHARE MODE does.

`LOCK TABLE jobs IN SHARE MODE` prevents other users from changing jobs, but not from locking jobs in share mode themselves. Changes to jobs will probably be much rarer than changes to employees. Therefore, locking jobs in share mode provides more concurrency than locking a single row of jobs in exclusive mode.

> **See Also:**
>
> - *Oracle Database Development Guide* for information about locking tables IN SHARE MODE
> - *Oracle Database PL/SQL Language Reference* for information about SELECT FOR UPDATE
> - "Creating Triggers"
> - "Tutorial: Showing How the employees_pkg Subprograms Work" to see how the employees_aiufer trigger works

Creating the Trigger to Enforce the Second Business Rule

The second business rule is: If an employee with job *j* has salary *s*, then you cannot change the minimum salary for *j* to a value greater than *s* or the maximum salary for *j* to a value less than *s*. (To do so would make existing data invalid.)

This rule could be violated when the min_salary or max_salary column of the jobs table is updated.

To enforce the rule, create the following trigger on the editioning view jobs. You can enter the CREATE TRIGGER statement either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the trigger with the SQL Developer tool Create Trigger.

```
CREATE OR REPLACE TRIGGER jobs_aufer
AFTER UPDATE OF min_salary, max_salary ON jobs FOR EACH ROW
WHEN (NEW.min_salary > OLD.min_salary OR NEW.max_salary < OLD.max_salary)
DECLARE
  l_cnt NUMBER;
BEGIN
  LOCK TABLE employees IN SHARE MODE;

  SELECT COUNT(*) INTO l_cnt
  FROM employees
  WHERE job_id = :NEW.job_id
  AND salary NOT BETWEEN :NEW.min_salary and :NEW.max_salary;

  IF (l_cnt>0) THEN
    RAISE_APPLICATION_ERROR( -20001,
      'Salary update would violate ' || l_cnt || ' existing employee records' );
  END IF;
END;
/
```

`LOCK TABLE employees IN SHARE MODE` prevents other users from changing the table employees while the trigger is querying it. Preventing changes to employees during the query is necessary because nonblocking reads prevent the trigger from "seeing"

changes that other users make to employees while the trigger is changing jobs (and prevent those users from "seeing" the changes that the trigger makes to jobs).

For this trigger, SELECT FOR UPDATE is not an alternative to LOCK TABLE IN SHARE MODE. While you are trying to change the salary range for this job, this trigger must prevent other users from changing a salary to be outside the new range. Therefore, the trigger must lock all rows in the employees table that have this job_id *and* lock all rows that someone could update to have this job_id.

One alternative to `LOCK TABLE employees IN SHARE MODE` is to use the DBMS_LOCK package to create a named lock with the name of the job_id and then use triggers on both the employees and jobs tables to use this named lock to prevent concurrent updates. However, using DBMS_LOCK and multiple triggers negatively impacts runtime performance.

Another alternative to `LOCK TABLE employees IN SHARE MODE` is to create a trigger on the employees table which, for each changed row of employees, locks the corresponding job row in jobs. However, this approach causes excessive work on updates to the employees table, which are frequent.

`LOCK TABLE employees IN SHARE MODE` is simpler than the preceding alternatives, and changes to the jobs table are rare and likely to happen at application maintenance time, when locking the table does not inconvenience users.

See Also:

- *Oracle Database Development Guide* for information about locking tables with `SHARE MODE`
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_LOCK` package
- "Creating Triggers"
- "Tutorial: Showing How the admin_pkg Subprograms Work"

Creating the Sequences

Note:

You must be connected to Oracle Database as user app_data.

To create the sequences that generate unique primary keys for new departments and new employees, use the following statements (in either order). You can enter the statements either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the sequences with the SQL Developer tool Create Sequence.

```
CREATE SEQUENCE employees_sequence START WITH 210;
CREATE SEQUENCE departments_sequence START WITH 275;
```

To avoid conflict with the data that you will load from tables in the sample schema HR, the starting numbers for employees_sequence and departments_sequence must exceed the

maximum values of employees.employee_id and departments.department_id, respectively. After "Loading the Data", this query displays these maximum values:

```
SELECT MAX(e.employee_id), MAX(d.department_id)
FROM employees e, departments d;
```

Result:

```
MAX(E.EMPLOYEE_ID) MAX(D.DEPARTMENT_ID)
------------------ --------------------
               206                  270
```

> **See Also:**
>
> "Creating and Managing Sequences"

Loading the Data

> **Note:**
>
> You must be connected to Oracle Database as user app_data.

Load the tables of the application with data from tables in the sample schema HR.

> **Note:**
>
> The following procedure references the tables of the application through their editioning views.

In the following procedure, you can enter the statements either in SQL*Plus or in the Worksheet of SQL Developer.

To load data into the tables:

1. Load jobs with data from the table HR.JOBS:

   ```
   INSERT INTO jobs (job_id, job_title, min_salary, max_salary)
   SELECT job_id, job_title, min_salary, max_salary
     FROM HR.JOBS
   /
   ```

   Result:

   ```
   19 rows created.
   ```

2. Load departments with data from the table HR.DEPARTMENTS:

   ```
   INSERT INTO departments (department_id, department_name, manager_id)
   SELECT department_id, department_name, manager_id
     FROM HR.DEPARTMENTS
   /
   ```

Result:

```
27 rows created.
```

3. Load employees with data from the tables HR.EMPLOYEES and HR.JOB_HISTORY, using searched CASE expressions and SQL functions to get employees.country_code and employees.phone_number from HR.phone_number and SQL functions and a scalar subquery to get employees.job_start_date from HR.JOB_HISTORY:

```
INSERT INTO employees (employee_id, first_name, last_name, email_addr,
  hire_date, country_code, phone_number, job_id, job_start_date, salary,
  manager_id, department_id)
SELECT employee_id, first_name, last_name, email, hire_date,
  CASE WHEN phone_number LIKE '011.%'
    THEN '+' || SUBSTR( phone_number, INSTR( phone_number, '.' )+1,
      INSTR( phone_number, '.', 1, 2 ) -  INSTR( phone_number, '.' ) - 1 )
    ELSE '+1'
  END country_code,
  CASE WHEN phone_number LIKE '011.%'
    THEN SUBSTR( phone_number, INSTR(phone_number, '.', 1, 2 )+1 )
    ELSE phone_number
  END phone_number,
  job_id,
  NVL( (SELECT MAX(end_date+1)
        FROM HR.JOB_HISTORY jh
        WHERE jh.employee_id = employees.employee_id), hire_date),
  salary, manager_id, department_id
  FROM HR.EMPLOYEES
/
```

Result:

```
107 rows created.
```

> **Note:**
>
> The preceding `INSERT` statement fires the trigger created in "Creating the Trigger to Enforce the First Business Rule".

4. Load job_history with data from the table HR.JOB_HISTORY:

```
INSERT INTO job_history (employee_id, job_id, start_date, end_date,
  department_id)
SELECT employee_id, job_id, start_date, end_date, department_id
  FROM HR.JOB_HISTORY
/
```

Result:

```
10 rows created.
```

5. Commit the changes:

```
COMMIT;
```

> **See Also:**
>
> - "About the INSERT Statement"
> - "About Sample Schema HR"
> - "Using CASE Expressions in Queries"
> - "Using NULL-Related Functions in Queries" for information about the `NVL` function
> - *Oracle Database SQL Language Reference* for information about the `SQL` functions

Adding the Foreign Key Constraint

> **Note:**
>
> You must be connected to Oracle Database as user app_data.

Now that the tables departments and employees contain data, add a foreign key constraint with the following ALTER TABLE statement. You can enter the statement either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can add the constraint with the SQL Developer tool Add Foreign Key.

```
ALTER TABLE departments#
ADD CONSTRAINT dept_to_emp_fk
FOREIGN KEY(manager_id) REFERENCES employees#;
```

If you add this foreign key constraint before departments# and employees# contain data, then you get this error when you try to load either of them with data:

```
ORA-02291: integrity constraint (APP_DATA.JOB_HIST_TO_DEPT_FK) violated - parent key not found
```

> **See Also:**
>
> "Tutorial: Adding Constraints to Existing Tables"

Granting Privileges on the Schema Objects to Users

> **Note:**
>
> You must be connected to Oracle Database as user app_data.

To grant privileges to users, use the SQL statement GRANT. You can enter the GRANT statements either in SQL*Plus or in the Worksheet of SQL Developer.

Grant to app_code *only* the privileges that it needs to create employees_pkg:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO app_code;
GRANT SELECT ON departments TO app_code;
GRANT SELECT ON jobs TO app_code;
GRANT SELECT, INSERT on job_history TO app_code;
GRANT SELECT ON employees_sequence TO app_code;
```

Grant to app_admin *only* the privileges that it needs to create admin_pkg:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON jobs TO app_admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON departments TO app_admin;
GRANT SELECT ON employees_sequence TO app_admin;
GRANT SELECT ON departments_sequence TO app_admin;
```

See Also:

Oracle Database SQL Language Reference for information about the `GRANT` statement

Creating the employees_pkg Package

This section shows how to create the employees_pkg package, how its subprograms work, how to grant the `EXECUTE` privilege on the package to the users who need it, and how those users can invoke one of its subprograms.

To create the employees_pkg package:

1. Connect to Oracle Database as user app_code.

   For instructions, see either "Connecting to Oracle Database from SQL*Plus" or "Connecting to Oracle Database from SQL Developer".

2. Create these synonyms:

   ```
   CREATE OR REPLACE SYNONYM employees FOR app_data.employees;
   CREATE OR REPLACE SYNONYM departments FOR app_data.departments;
   CREATE OR REPLACE SYNONYM jobs FOR app_data.jobs;
   CREATE OR REPLACE SYNONYM job_history FOR app_data.job_history;
   ```

   You can enter the CREATE SYNONYM statements either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the synonyms with the SQL Developer tool Create Synonym.

3. Create the package specification.
4. Create the package body.

See Also:

- "Creating Synonyms"
- "About Packages"

Creating the Package Specification for employees_pkg

> **Note:**
>
> You must be connected to Oracle Database as user app_code.

To create the package specification for employees_pkg, the API for managers, use the following CREATE PACKAGE statement. You can enter the statement either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the package with the SQL Developer tool Create Package.

```
CREATE OR REPLACE PACKAGE employees_pkg
AS
  PROCEDURE get_employees_in_dept
    ( p_deptno     IN     employees.department_id%TYPE,
      p_result_set IN OUT SYS_REFCURSOR );

  PROCEDURE get_job_history
    ( p_employee_id  IN     employees.department_id%TYPE,
      p_result_set   IN OUT SYS_REFCURSOR );

  PROCEDURE show_employee
    ( p_employee_id  IN     employees.employee_id%TYPE,
      p_result_set   IN OUT SYS_REFCURSOR );

  PROCEDURE update_salary
    ( p_employee_id IN employees.employee_id%TYPE,
      p_new_salary  IN employees.salary%TYPE );

  PROCEDURE change_job
    ( p_employee_id IN employees.employee_id%TYPE,
      p_new_job     IN employees.job_id%TYPE,
      p_new_salary  IN employees.salary%TYPE := NULL,
      p_new_dept    IN employees.department_id%TYPE := NULL );

END employees_pkg;
/
```

> **See Also:**
>
> - "About the Application"
> - "Creating and Managing Packages"
> - *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE statement

Creating the Package Body for employees_pkg

> **Note:**
>
> You must be connected to Oracle Database as user app_code.

To create the package body for employees_pkg, the API for managers, use the following CREATE PACKAGE BODY statement. You can enter the statement either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the package with the SQL Developer tool Create Body.

```
CREATE OR REPLACE PACKAGE BODY employees_pkg
AS
  PROCEDURE get_employees_in_dept
    ( p_deptno     IN     employees.department_id%TYPE,
      p_result_set IN OUT SYS_REFCURSOR )
  IS
     l_cursor SYS_REFCURSOR;
  BEGIN
    OPEN p_result_set FOR
      SELECT e.employee_id,
        e.first_name || ' ' || e.last_name name,
        TO_CHAR( e.hire_date, 'Dy Mon ddth, yyyy' ) hire_date,
        j.job_title,
        m.first_name || ' ' || m.last_name manager,
        d.department_name
      FROM employees e INNER JOIN jobs j ON (e.job_id = j.job_id)
        LEFT OUTER JOIN employees m ON (e.manager_id = m.employee_id)
        INNER JOIN departments d ON (e.department_id = d.department_id)
      WHERE e.department_id = p_deptno ;
  END get_employees_in_dept;

  PROCEDURE get_job_history
    ( p_employee_id  IN     employees.department_id%TYPE,
      p_result_set   IN OUT SYS_REFCURSOR )
  IS
  BEGIN
    OPEN p_result_set FOR
      SELECT e.First_name || ' ' || e.last_name name, j.job_title,
        e.job_start_date start_date,
        TO_DATE(NULL) end_date
      FROM employees e INNER JOIN jobs j ON (e.job_id = j.job_id)
      WHERE e.employee_id = p_employee_id
      UNION ALL
      SELECT e.First_name || ' ' || e.last_name name,
        j.job_title,
        jh.start_date,
        jh.end_date
      FROM employees e INNER JOIN job_history jh
        ON (e.employee_id = jh.employee_id)
        INNER JOIN jobs j ON (jh.job_id = j.job_id)
      WHERE e.employee_id = p_employee_id
      ORDER BY start_date DESC;
  END get_job_history;
```

```
  PROCEDURE show_employee
    ( p_employee_id  IN     employees.employee_id%TYPE,
      p_result_set   IN OUT sys_refcursor )
  IS
  BEGIN
    OPEN p_result_set FOR
      SELECT *
      FROM (SELECT TO_CHAR(e.employee_id) employee_id,
              e.first_name || ' ' || e.last_name name,
              e.email_addr,
              TO_CHAR(e.hire_date,'dd-mon-yyyy') hire_date,
              e.country_code,
              e.phone_number,
              j.job_title,
              TO_CHAR(e.job_start_date,'dd-mon-yyyy') job_start_date,
              to_char(e.salary) salary,
              m.first_name || ' ' || m.last_name manager,
              d.department_name
            FROM employees e INNER JOIN jobs j on (e.job_id = j.job_id)
              RIGHT OUTER JOIN employees m ON (m.employee_id = e.manager_id)
              INNER JOIN departments d ON (e.department_id = d.department_id)
            WHERE e.employee_id = p_employee_id)
      UNPIVOT (VALUE FOR ATTRIBUTE IN (employee_id, name, email_addr, hire_date,
        country_code, phone_number, job_title, job_start_date, salary, manager,
        department_name) );
  END show_employee;

  PROCEDURE update_salary
    ( p_employee_id IN employees.employee_id%type,
      p_new_salary  IN employees.salary%type )
  IS
  BEGIN
    UPDATE employees
    SET salary = p_new_salary
    WHERE employee_id = p_employee_id;
  END update_salary;

  PROCEDURE change_job
    ( p_employee_id IN employees.employee_id%TYPE,
      p_new_job     IN employees.job_id%TYPE,
      p_new_salary  IN employees.salary%TYPE := NULL,
      p_new_dept    IN employees.department_id%TYPE := NULL )
  IS
  BEGIN
    INSERT INTO job_history (employee_id, start_date, end_date, job_id,
      department_id)
    SELECT employee_id, job_start_date, TRUNC(SYSDATE), job_id, department_id
      FROM employees
      WHERE employee_id = p_employee_id;

    UPDATE employees
    SET job_id = p_new_job,
      department_id = NVL( p_new_dept, department_id ),
      salary = NVL( p_new_salary, salary ),
      job_start_date = TRUNC(SYSDATE)
    WHERE employee_id = p_employee_id;
  END change_job;
END employees_pkg;
/
```

> **See Also:**
>
> - "About the Application"
> - "Creating and Managing Packages"
> - *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE BODY statement

Tutorial: Showing How the employees_pkg Subprograms Work

Using SQL*Plus, this tutorial shows how the subprograms of the employees_pkg package work. The tutorial also shows how the trigger employees_aiufer and the CHECK constraint job_history_date_check work.

> **Note:**
>
> You must be connected to Oracle Database as user `app_code` from SQL*Plus.

To use SQL*Plus to show how the employees_pkg subprograms work:

1. Use formatting commands to improve the readability of the output. For example:

```
SET LINESIZE 80
SET RECSEP WRAPPED
SET RECSEPCHAR "="
COLUMN NAME FORMAT A15 WORD_WRAPPED
COLUMN HIRE_DATE FORMAT A20 WORD_WRAPPED
COLUMN DEPARTMENT_NAME FORMAT A10 WORD_WRAPPED
COLUMN JOB_TITLE FORMAT A29 WORD_WRAPPED
COLUMN MANAGER FORMAT A11 WORD_WRAPPED
```

2. Declare a bind variable for the value of the subprogram parameter `p_result_set`:

```
VARIABLE c REFCURSOR
```

3. Show the employees in department 90:

```
EXEC employees_pkg.get_employees_in_dept( 90, :c );
PRINT c
```

Result:

```
EMPLOYEE_ID NAME            HIRE_DATE            JOB_TITLE
----------- --------------- -------------------- --------------------------
MANAGER     DEPARTMENT
----------- ----------
        100 Steven King     Tue Jun 17th, 2003   President
            Executive
==============================================================================
        102 Lex De Haan     Sat Jan 13th, 2001   Administration Vice President
Steven King Executive
==============================================================================
        101 Neena Kochhar   Wed Sep 21st, 2005   Administration Vice President
```

```
Steven King Executive
================================================================================
```

4. Show the job history of employee 101:

```
EXEC employees_pkg.get_job_history( 101, :c );
PRINT c
```

Result:

```
NAME            JOB_TITLE                     START_DAT END_DATE
--------------- ----------------------------- --------- ---------
Neena Kochhar   Administration Vice President 16-MAR-05
Neena Kochhar   Accounting Manager            28-OCT-01 15-MAR-05
Neena Kochhar   Public Accountant             21-SEP-97 27-OCT-01
```

5. Show general information about employee 101:

```
EXEC employees_pkg.show_employee( 101, :c );
PRINT c
```

Result:

```
ATTRIBUTE       VALUE
--------------- ----------------------------------------------
EMPLOYEE_ID     101
NAME            Neena Kochhar
EMAIL_ADDR      NKOCHHAR
HIRE_DATE       21-sep-2005
COUNTRY_CODE    +1
PHONE_NUMBER    515.123.4568
JOB_TITLE       Administration Vice President
JOB_START_DATE  16-mar-05
SALARY          17000
MANAGER         Steven King
DEPARTMENT_NAME Executive

11 rows selected.
```

6. Show the information about the job Administration Vice President:

```
SELECT * FROM jobs WHERE job_title = 'Administration Vice President';
```

Result:

```
JOB_ID     JOB_TITLE                     MIN_SALARY MAX_SALARY
---------- ----------------------------- ---------- ----------
AD_VP      Administration Vice President      15000      30000
```

7. Try to give employee 101 a new salary outside the range for their job:

```
EXEC employees_pkg.update_salary( 101, 30001 );
```

Result:

```
SQL> EXEC employees_pkg.update_salary( 101, 30001 );
BEGIN employees_pkg.update_salary( 101, 30001 ); END;

*
ERROR at line 1:
ORA-20002: Salary modification invalid
ORA-06512: at "APP_DATA.EMPLOYEES_AIUFER", line 13
ORA-04088: error during execution of trigger 'APP_DATA.EMPLOYEES_AIUFER'
```

```
ORA-06512: at "APP_CODE.EMPLOYEES_PKG", line 77
ORA-06512: at line 1
```

8. Give employee 101 a new salary inside the range for their job and show general information about them again:

```
EXEC employees_pkg.update_salary( 101, 18000 );
EXEC employees_pkg.show_employee( 101, :c );
PRINT c
```

Result:

```
ATTRIBUTE       VALUE
--------------- ----------------------------------------------
EMPLOYEE_ID     101
NAME            Neena Kochhar
EMAIL_ADDR      NKOCHHAR
HIRE_DATE       21-sep-2005
COUNTRY_CODE    +1
PHONE_NUMBER    515.123.4568
JOB_TITLE       Administration Vice President
JOB_START_DATE  16-mar-05
SALARY          18000
MANAGER         Steven King
DEPARTMENT_NAME Executive

11 rows selected.
```

9. Change the job of employee 101 to their current job with a lower salary:

```
EXEC employees_pkg.change_job( 101, 'AD_VP', 17500, 90 );
```

Result:

```
SQL> exec employees_pkg.change_job( 101, 'AD_VP', 17500, 90 );
BEGIN employees_pkg.change_job( 101, 'AD_VP', 17500, 80 ); END;

*
ERROR at line 1:
ORA-02290: check constraint (APP_DATA.JOB_HISTORY_DATE_CHECK) violated
ORA-06512: at "APP_CODE.EMPLOYEES_PKG", line 101
ORA-06512: at line 1
```

10. Show information about the employee. (Note that the salary was not changed by the statement in the preceding step; it is 18000, not 17500.)

```
exec employees_pkg.show_employee( 101, :c );
print c
```

Result:

```
ATTRIBUTE       VALUE
--------------- ----------------------------------------------
EMPLOYEE_ID     101
NAME            Neena Kochhar
EMAIL_ADDR      NKOCHHAR
HIRE_DATE       21-sep-2005
COUNTRY_CODE    +1
PHONE_NUMBER    515.123.4568
JOB_TITLE       Administration Vice President
JOB_START_DATE  10-mar-2015
SALARY          18000
MANAGER         Steven King
DEPARTMENT_NAME Executive
```

```
11 rows selected.
```

> **See Also:**
>
> - *SQL*Plus User's Guide and Reference* for information about SQL*Plus commands
> - "Creating and Managing Packages"

Granting the EXECUTE Privilege to app_user and app_admin_user

> **Note:**
>
> You must be connected to Oracle Database as user app_code.

To grant the `EXECUTE` privilege on the package employees_pkg to app_user (typically a manager) and app_admin_user (an application administrator), use the following GRANT statements (in either order). You can enter the statements either in SQL*Plus or in the Worksheet of SQL Developer.

```
GRANT EXECUTE ON employees_pkg TO app_user;
GRANT EXECUTE ON employees_pkg TO app_admin_user;
```

> **See Also:**
>
> - "Schemas for the Application"
> - *Oracle Database SQL Language Reference* for information about the GRANT statement

Tutorial: Invoking get_job_history as app_user or app_admin_user

Using SQL*Plus, this tutorial shows how to invoke the subprogram app_code.employees_pkg.get_job_history as the user app_user (typically a manager) or app_admin_user (an application administrator).

To invoke employees_pkg.get_job_history as app_user or app_admin_user:

1. Connect to Oracle Database as user app_user or app_admin_user from SQL*Plus.

   For instructions, see "Connecting to Oracle Database from SQL*Plus".

2. Create this synonym:

   ```
   CREATE SYNONYM employees_pkg FOR app_code.employees_pkg;
   ```

3. Show the job history of employee 101:

```
EXEC employees_pkg.get_job_history( 101, :c );
PRINT c
```

Result:

```
NAME            JOB_TITLE                     START_DAT END_DATE
--------------- ----------------------------- --------- ---------
Neena Kochhar   Administration Vice President 16-MAR-05 15-MAY-12
Neena Kochhar   Accounting Manager            28-OCT-01 15-MAR-05
Neena Kochhar   Public Accountant             21-SEP-97 27-OCT-01
```

Creating the admin_pkg Package

This section shows how to create the admin_pkg package, how its subprograms work, how to grant the `EXECUTE` privilege on the package to the user who needs it, and how that user can invoke one of its subprograms.

To create the admin_pkg package:

1. Connect to Oracle Database as user app_admin.

   For instructions, see either "Connecting to Oracle Database from SQL*Plus" or "Connecting to Oracle Database from SQL Developer".

2. Create these synonyms:

   ```
   CREATE SYNONYM departments FOR app_data.departments;
   CREATE SYNONYM jobs FOR app_data.jobs;
   CREATE SYNONYM departments_sequence FOR app_data.departments_sequence;
   ```

   You can enter the CREATE SYNONYM statements either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the tables with the SQL Developer tool Create Synonym.

3. Create the package specification.
4. Create the package body.

> **See Also:**
>
> - "Creating and Managing Synonyms"
> - "About Packages"

Creating the Package Specification for admin_pkg

> **Note:**
>
> You must be connected to Oracle Database as user app_admin.

To create the package specification for admin_pkg, the API for application administrators, use the following CREATE PACKAGE statement. You can enter the statement either in SQL*Plus

or in the Worksheet of SQL Developer. Alternatively, you can create the package with the SQL Developer tool Create Package.

```
CREATE OR REPLACE PACKAGE admin_pkg
AS
  PROCEDURE update_job
    ( p_job_id      IN jobs.job_id%TYPE,
      p_job_title   IN jobs.job_title%TYPE := NULL,
      p_min_salary  IN jobs.min_salary%TYPE := NULL,
      p_max_salary  IN jobs.max_salary%TYPE := NULL );

  PROCEDURE add_job
    ( p_job_id      IN jobs.job_id%TYPE,
      p_job_title   IN jobs.job_title%TYPE,
      p_min_salary  IN jobs.min_salary%TYPE,
      p_max_salary  IN jobs.max_salary%TYPE );

  PROCEDURE update_department
    ( p_department_id     IN departments.department_id%TYPE,
      p_department_name   IN departments.department_name%TYPE := NULL,
      p_manager_id        IN departments.manager_id%TYPE := NULL,
      p_update_manager_id IN BOOLEAN := FALSE );

  FUNCTION add_department
    ( p_department_name   IN departments.department_name%TYPE,
      p_manager_id        IN departments.manager_id%TYPE )
    RETURN departments.department_id%TYPE;

END admin_pkg;
/
```

See Also:

- "About the Application"
- "Creating and Managing Packages"
- *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE statement

Creating the Package Body for admin_pkg

Note:

You must be connected to Oracle Database as user app_admin.

To create the package body for admin_pkg, the API for application administrators, use the following CREATE PACKAGE BODY statement. You can enter the statement either in SQL*Plus or in the Worksheet of SQL Developer. Alternatively, you can create the package with the SQL Developer tool Create Body.

```
CREATE OR REPLACE PACKAGE BODY admin_pkg
AS
```

```
  PROCEDURE update_job
    ( p_job_id      IN jobs.job_id%TYPE,
      p_job_title   IN jobs.job_title%TYPE := NULL,
      p_min_salary  IN jobs.min_salary%TYPE := NULL,
      p_max_salary  IN jobs.max_salary%TYPE := NULL )
  IS
  BEGIN
    UPDATE jobs
    SET job_title  = NVL( p_job_title, job_title ),
        min_salary = NVL( p_min_salary, min_salary ),
        max_salary = NVL( p_max_salary, max_salary )
    WHERE job_id = p_job_id;
  END update_job;

  PROCEDURE add_job
    ( p_job_id      IN jobs.job_id%TYPE,
      p_job_title   IN jobs.job_title%TYPE,
      p_min_salary  IN jobs.min_salary%TYPE,
      p_max_salary  IN jobs.max_salary%TYPE )
  IS
  BEGIN
    INSERT INTO jobs ( job_id, job_title, min_salary, max_salary )
    VALUES ( p_job_id, p_job_title, p_min_salary, p_max_salary );
  END add_job;

  PROCEDURE update_department
    ( p_department_id     IN departments.department_id%TYPE,
      p_department_name   IN departments.department_name%TYPE := NULL,
      p_manager_id        IN departments.manager_id%TYPE := NULL,
      p_update_manager_id IN BOOLEAN := FALSE )
  IS
  BEGIN
    IF ( p_update_manager_id ) THEN
      UPDATE departments
      SET department_name = NVL( p_department_name, department_name ),
          manager_id = p_manager_id
      WHERE department_id = p_department_id;
    ELSE
      UPDATE departments
      SET department_name = NVL( p_department_name, department_name )
      WHERE department_id = p_department_id;
    END IF;
  END update_department;

  FUNCTION add_department
    ( p_department_name   IN departments.department_name%TYPE,
      p_manager_id        IN departments.manager_id%TYPE )
    RETURN departments.department_id%TYPE
  IS
    l_department_id departments.department_id%TYPE;
  BEGIN
    INSERT INTO departments ( department_id, department_name, manager_id )
      VALUES ( departments_sequence.NEXTVAL, p_department_name, p_manager_id )
      RETURNING department_id INTO l_department_id;
    RETURN l_department_id;
  END add_department;

END admin_pkg;
/
```

> **See Also:**
>
> - "About the Application"
> - "Creating and Managing Packages"
> - *Oracle Database PL/SQL Language Reference* for information about the CREATE PACKAGE BODY statement

Tutorial: Showing How the admin_pkg Subprograms Work

Using SQL*Plus, this tutorial shows how the subprograms of the admin_pkg package work. The tutorial also shows how the trigger jobs_aufer works.

> **Note:**
>
> You must be connected to Oracle Database as user app_admin from SQL*Plus.

To show how the admin_pkg subprograms work:

1. Show the information about the job whose ID is AD_VP:

```
SELECT * FROM jobs WHERE job_id = 'AD_VP';
```

Result:

```
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
AD_VP      Administration Vice President           15000      30000
```

2. Increase the maximum salary for this job and show the information about it again:

```
EXEC admin_pkg.update_job( 'AD_VP', p_max_salary => 31000 );
SELECT * FROM jobs WHERE job_id = 'AD_VP';
```

Result:

```
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
AD_VP      Administration Vice President           15000      31000
```

3. Show the information about the job whose ID is IT_PROG:

```
SELECT * FROM jobs WHERE job_id = 'IT_PROG';
```

Result:

```
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
IT_PROG    Programmer                               4000      10000
```

4. Try to increase the maximum salary for this job:

```
EXEC admin_pkg.update_job( 'IT_PROG', p_max_salary => 4001 );
```

Result (from SQL*Plus):

```
SQL> EXEC admin_pkg.update_job( 'IT_PROG', p_max_salary => 4001 );
BEGIN admin_pkg.update_job( 'IT_PROG', p_max_salary => 4001 ); END;

*
ERROR at line 1:
ORA-20001: Salary update would violate 5 existing employee records
ORA-06512: at "APP_DATA.JOBS_AUFER", line 12
ORA-04088: error during execution of trigger 'APP_DATA.JOBS_AUFER'
ORA-06512: at "APP_ADMIN.ADMIN_PKG", line 10
ORA-06512: at line 1
```

5. Add a new job and show the information about it:

```
EXEC admin_pkg.add_job( 'AD_CLERK', 'Administrative Clerk', 3000, 7000 );
SELECT * FROM jobs WHERE job_id = 'AD_CLERK';
```

Result:

```
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
AD_CLERK   Administrative Clerk                     3000       7000
```

6. Show the information about department 100:

```
SELECT * FROM departments WHERE department_id = 100;
```

Result:

```
DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID
------------- ------------------------------ ----------
          100 Finance                               108
```

7. Change the name and manager of department 100 and show the information about it:

```
EXEC admin_pkg.update_department( 100, 'Financial Services' );
EXEC admin_pkg.update_department( 100, p_manager_id => 111,
  p_update_manager_id => true );
SELECT * FROM departments WHERE department_id = 100;
```

Result:

```
DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID
------------- ------------------------------ ----------
          100 Financial Services                    111
```

> **See Also:**
>
> "Creating and Managing Packages"

Granting the EXECUTE Privilege to app_admin_user

> **Note:**
>
> You must be connected to Oracle Database as user app_admin.

To grant the `EXECUTE` privilege on the package admin_pkg to app_admin_user (an application administrator), use the following GRANT statement. You can enter the statement either in SQL*Plus or in the Worksheet of SQL Developer.

```
GRANT EXECUTE ON admin_pkg TO app_admin_user;
```

> **See Also:**
>
> - "Schemas for the Application"
> - *Oracle Database SQL Language Reference* for information about the GRANT statement

Tutorial: Invoking add_department as app_admin_user

Using SQL*Plus, this tutorial shows how to invoke the function app_admin.admin_pkg.add_department as the user app_admin_user (an application administrator) and then see the information about the new department.

To invoke admin_pkg.add_department as app_admin_user:

1. Connect to Oracle Database as user app_admin_user from SQL*Plus.

   For instructions, see "Connecting to Oracle Database from SQL*Plus".

2. Create this synonym:

   ```
   CREATE SYNONYM admin_pkg FOR app_admin.admin_pkg;
   ```

3. Declare a bind variable for the return value of the function:

   ```
   VARIABLE n NUMBER
   ```

4. Add a new department without a manager:

   ```
   EXEC :n := admin_pkg.add_department( 'New department', NULL );
   ```

5. Show the ID of the manager of the new department:

   ```
   PRINT :n
   ```

   Result:

   ```
            N
   ----------
          275
   ```

To see the information about the new department:

1. Connect to Oracle Database as user app_admin.
2. Show the information about the new department:

   ```
   SELECT * FROM departments WHERE department_name LIKE 'New department%';
   ```

   Result:

   ```
   DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID
   ------------- ------------------------------ ----------
             275 New department
   ```