Fidelity
INVESTMENTS

WORKING WITH RELATIONAL DATABASES

EXERCISE MANUAL

ROITRAINING
MAXIMIZE YOUR TRAINING INVESTMENT

Fidelity LEAP
Technology Immersion Program

This page was intentionally left blank.

Table of Contents

Chapter 1: What Is Structured Query Language?

Exercise 1.1: Using SQL Developer

For this exercise, we will become familiar with SQL Developer.

1. Locate the SQL Developer executable on the desktop.
2. Double-click to see that it starts correctly.
3. Right-click the `HR` connection and select Properties from the menu. Examine the parameters. Where is the Oracle database server running?
4. Make the following configuration changes in the Tools | Preferences Menu:
   a. In the Code Editor Line Gutter section, choose the option to **Show Line Numbers**.

      Tools | Preferences | Code Editor | Line Gutter

   b. In the Database NLS Parameters section, modify the Date Format to display **DD-MON-YYYY**.

      Tools | Preferences | Database | NLS Parameters
5. Open the `HR` connection.

   You can do this either by clicking the + (plus sign) to the left of the Connection Name or by right-clicking the Connection icon and selecting Connect.
6. Expand the **Tables** branch to show all of the tables from `HR`.

   Click the + (plus sign) or double-click the name.
7. How many tables are owned by `HR`?

   ____

8. Open SQL Worksheet for `HR`.

   Use the SQL Worksheet icon in the toolbar.
9. Enter the following statement into the SQL Worksheet window and execute it:

   ```
   SELECT * FROM locations;
   ```

   Click the Run Statement icon or the <F9> key to run the statement.

10. Execute the same statement as a script and note the difference.

    Click the Run Script icon or the <F5> key to run the script.

11. Save the SQL script for later use as a file called `loc.sql`.

12. Enter the following statement underneath the existing statement.

    ```
    SELECT * FROM countries;
    ```

13. Execute both statements by clicking the <F9> key and note the result.

14. Now, execute both statements together as a script (<F5>). Note that the results of both statements are displayed in the Script Output.

15. Clear the contents of the SQL Worksheet.

16. Open the `loc.sql` file.

    Click the Open File icon or use the File menu and select Open.

17. Run the SQL statement.

Congratulations! You have completed this exercise.

Chapter 2: SQL Query Syntax

Exercise 2.1: Selecting Data

Connect to the SCOTT account

Unless the order is specified, the order of your results may differ.

1. Write a query to display the dname and deptno of all rows in the dept table.

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

2. Write a query to display ALL of the columns and rows in the dept table.

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

3. Write a query to display the dname, deptno, and location of all rows in the dept table labeling them Name, DEPT# and Dept Location, respectively.

```
Name                DEPT# Dept Location
-------------- ---------- -------------
ACCOUNTING             10 NEW YORK
RESEARCH               20 DALLAS
SALES                  30 CHICAGO
OPERATIONS             40 BOSTON
```

4. Write a query to display the deptno of each row in the emp table.

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

5. Write a query to display each `deptno` in the `emp` table only once.

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

6. Write a query to display the `deptno` and `job` of each row in the `emp` table.

```
    DEPTNO JOB
---------- ---------
        20 CLERK
        30 SALESMAN
        30 SALESMAN
        20 MANAGER
        30 SALESMAN
        30 MANAGER
        10 MANAGER
        20 ANALYST
        10 PRESIDENT
        30 SALESMAN
        20 CLERK
        30 CLERK
        20 ANALYST
        10 CLERK
```

7. Write a query to display each unique combination of `deptno` and `job` in the `emp` table.

```
    DEPTNO JOB
---------- ---------
        20 MANAGER
        20 ANALYST
        10 PRESIDENT
        10 CLERK
        30 SALESMAN
        10 MANAGER
        20 CLERK
        30 MANAGER
        30 CLERK
```

8. Write a query to display the names of the employees who work for `dept 30` in the `emp` table.

```
ENAME
----------
ALLEN
WARD
MARTIN
BLAKE
TURNER
JAMES
```

9. Write a query to display the names of the employees who were hired on Dec. 17, 1981. Specify the date in a safe format.

```
no rows selected
```

10. Write a query to display the names of the employees who were hired on or after Dec. 17, 1981.

```
ENAME
----------
SCOTT
ADAMS
MILLER
```

11. Write a query to display the names of the employees who have the job of clerk.

```
no rows selected
```

12. Write a query to display the names of the employees who have the job of CLERK.

```
ENAME
----------
SMITH
ADAMS
JAMES
MILLER
```

13. Write a query to display the names of the employees whose salary is greater than 2500.

```
ENAME
----------
JONES
BLAKE
SCOTT
KING
FORD
```

14. Write a query to display the names of the employees whose salary is in the range (inclusive) of 1000 and 1600.

```
NAME
----------
ALLEN
WARD
MARTIN
TURNER
ADAMS
MILLER
```

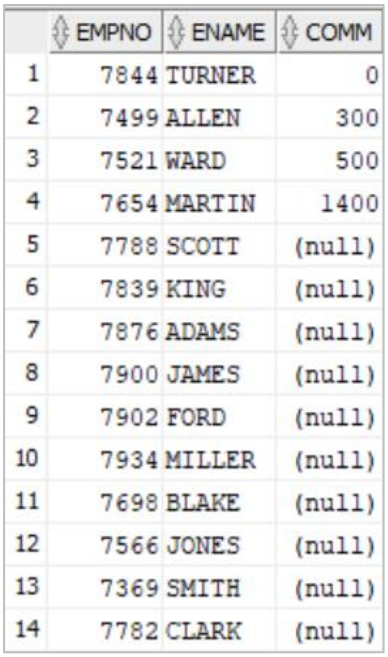
15. Write a query to display the names of the employees whose names contain "ER".

```
ENAME
----------
TURNER
MILLER
```

16. Write a query to display the names and employee numbers of the employees whose commission is undefined.

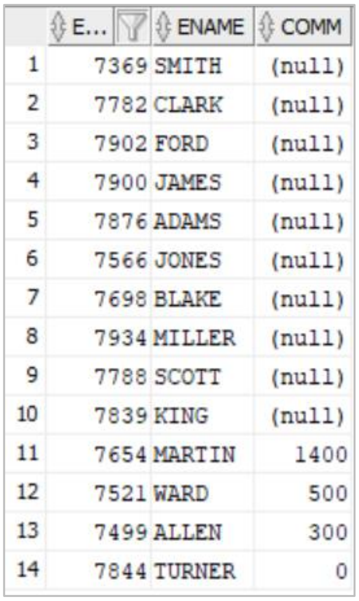
```
     EMPNO ENAME
---------- ----------
      7369 SMITH
      7566 JONES
      7698 BLAKE
      7782 CLARK
      7788 SCOTT
      7839 KING
      7876 ADAMS
      7900 JAMES
      7902 FORD
      7934 MILLER
```

17. Write a query to display the names, employee numbers, and commissions of the employees, sequencing the data in commission ascending order.



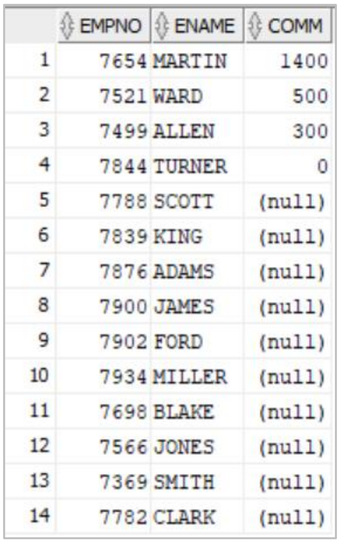
	EMPNO	ENAME	COMM
1	7844	TURNER	0
2	7499	ALLEN	300
3	7521	WARD	500
4	7654	MARTIN	1400
5	7788	SCOTT	(null)
6	7839	KING	(null)
7	7876	ADAMS	(null)
8	7900	JAMES	(null)
9	7902	FORD	(null)
10	7934	MILLER	(null)
11	7698	BLAKE	(null)
12	7566	JONES	(null)
13	7369	SMITH	(null)
14	7782	CLARK	(null)

18. Write a query to display the names, employee numbers, and commissions of the employees sequencing the data in commission descending order.



	E...	ENAME	COMM
1	7369	SMITH	(null)
2	7782	CLARK	(null)
3	7902	FORD	(null)
4	7900	JAMES	(null)
5	7876	ADAMS	(null)
6	7566	JONES	(null)
7	7698	BLAKE	(null)
8	7934	MILLER	(null)
9	7788	SCOTT	(null)
10	7839	KING	(null)
11	7654	MARTIN	1400
12	7521	WARD	500
13	7499	ALLEN	300
14	7844	TURNER	0

19. Write a query to display the names and employee numbers of the employees sequencing the data in commission descending order, forcing those with unknown commissions to the bottom of the list.



	EMPNO	ENAME	COMM
1	7654	MARTIN	1400
2	7521	WARD	500
3	7499	ALLEN	300
4	7844	TURNER	0
5	7788	SCOTT	(null)
6	7839	KING	(null)
7	7876	ADAMS	(null)
8	7900	JAMES	(null)
9	7902	FORD	(null)
10	7934	MILLER	(null)
11	7698	BLAKE	(null)
12	7566	JONES	(null)
13	7369	SMITH	(null)
14	7782	CLARK	(null)

Congratulations! You have completed this exercise.

This page intentionally left blank.

Chapter 3: SQL Scalar Functions

Exercise 3.1: Using Scalar Functions

Unless the order is specified, the order of your results may differ.

Connect to the `HR` account.

1. Write a query to display the first name. last name, and salary of all employees in department 30, formatting the salary with commas and a floating dollar sign.

```
FIRST_NAME           LAST_NAME                 SALARY
-------------------- ------------------------- --------
Den                  Raphaely                   $11,000
Alexander            Khoo                        $3,100
Shelli               Baida                       $2,900
Sigal                Tobias                      $2,800
Guy                  Himuro                      $2,600
Karen                Colmenares                  $2,500
```

2. Write a query to display the first name, last name, and date hired of all employees in department 30, formatting the date to be year-month#-day.

```
FIRST_NAME           LAST_NAME                 Date Hired
-------------------- ------------------------- ----------
Den                  Raphaely                  2002-12-07
Alexander            Khoo                      2003-05-18
Shelli               Baida                     2005-12-24
Sigal                Tobias                    2005-07-24
Guy                  Himuro                    2006-11-15
Karen                Colmenares                2007-08-10
```

3. Write a query to display the salary of all employees in department 30. Also show the salary rounded and truncated to thousands.

```
FIRST_NAME           LAST_NAME                   RDSAL    TSAL  SALARY
-------------------- ----------------------- ------- ------- -------
Den                  Raphaely                  11000   11000   11000
Alexander            Khoo                       3000    3000    3100
Shelli               Baida                      3000    2000    2900
Sigal                Tobias                     3000    2000    2800
Guy                  Himuro                     3000    2000    2600
Karen                Colmenares                 3000    2000    2500
```

4. Write a query to display names of all employees in department 30. Their first name should be in lower case; their last name in upper case. Sequence the list in (ascending) first name, last name order.

```
LNAME                UNAME
-------------------- ---------------
alexander            KHOO
den                  RAPHAELY
guy                  HIMURO
karen                COLMENARES
shelli               BAIDA
sigal                TOBIAS
```

5. Write a query to display the initial of the first name followed by a period followed by the last name of all employees in department 30. Sequence the list in alphabetical order of this formatted name.

```
NAME
--------------
A. Khoo
D. Raphaely
G. Himuro
K. Colmenares
S. Baida
S. Tobias
```

6. Write a query to display the street address, followed by the street address stripped of any leading numeric digits, spaces, or dashes (Street Name) for all rows in the `locations` table. Order the list by the Street Name.

```
STREET_ADDRESS                           Street Name
---------------------------------------- ----------------------------------------
8204 Arthur St                           Arthur St
6092 Boxwood St                          Boxwood St
93091 Calle della Testa                  Calle della Testa
2004 Charade Rd                          Charade Rd
9702 Chester Road                        Chester Road
198 Clementi North                       Clementi North
2011 Interiors Blvd                      Interiors Blvd
2014 Jabberwocky Rd                      Jabberwocky Rd
9450 Kamiya-cho                          Kamiya-cho
40-5-12 Laogianggen                      Laogianggen
Magdalen Centre, The Oxford Science Park Magdalen Centre, The Oxford Science Park
Mariano Escobedo 9991                    Mariano Escobedo 9991
Murtenstrasse 921                        Murtenstrasse 921
Pieter Breughelstraat 837                Pieter Breughelstraat 837
Rua Frei Caneca 1360                     Rua Frei Caneca 1360
20 Rue des Corps-Saints                  Rue des Corps-Saints
Schwanthalerstr. 7031                    Schwanthalerstr. 7031
2017 Shinjuku-ku                         Shinjuku-ku
147 Spadina Ave                          Spadina Ave
1297 Via Cola di Rie                     Via Cola di Rie
12-98 Victoria Street                    Victoria Street
1298 Vileparle (E)                       Vileparle (E)
2007 Zagora St                           Zagora St
```

7. Write a query to display the street address, followed by length of the street address (Street Length) for all rows in the `locations` table. Sequence the list in the Street Length order.

```
STREET_ADDRESS                           Street Length
---------------------------------------- -------------
8204 Arthur St                                      14
2007 Zagora St                                      14
2004 Charade Rd                                     15
9450 Kamiya-cho                                     15
6092 Boxwood St                                     15
147 Spadina Ave                                     15
2017 Shinjuku-ku                                    16
Murtenstrasse 921                                   17
9702 Chester Road                                   17
1298 Vileparle (E)                                  18
198 Clementi North                                  18
2014 Jabberwocky Rd                                 19
40-5-12 Laogianggen                                 19
2011 Interiors Blvd                                 19
1297 Via Cola di Rie                                20
Schwanthalerstr. 7031                               21
12-98 Victoria Street                               21
Rua Frei Caneca 1360                                21
Mariano Escobedo 9991                               21
20 Rue des Corps-Saints                             23
93091 Calle della Testa                             23
Pieter Breughelstraat 837                           25
Magdalen Centre, The Oxford Science Park            40
```

8. Write a query to display the location ID, the street address, city, and state province of all rows in the `locations` table that contain either the string "RUA" or "RUE" in the street address. Sequence the list in descending sequence on location ID.

```
LOCATION_ID STREET_ADDRESS            CITY          STATE_PROVINCE
----------- ------------------------- ------------- ---------------
       2900 20 Rue des Corps-Saints   Geneva        Geneve
       2800 Rua Frei Caneca 1360      Sao Paulo     Sao Paulo
```

Congratulations! You have completed this exercise.

This page intentionally left blank.

Chapter 4: SQL Joins

Exercise 4.1: Working with `INNER JOINS`

Save your queries for later use.

Connect to the `HR` account.

1. Joining the `locations` and `departments` tables, display the city, location ID, and department name.

```
CITY                         LOCATION_ID DEPARTMENT_NAME
---------------------------- ----------- --------------------
Southlake                           1400 IT
South San Francisco                 1500 Shipping
Seattle                             1700 Administration
Seattle                             1700 Purchasing
Seattle                             1700 Executive
Seattle                             1700 Finance
Seattle                             1700 Accounting
Seattle                             1700 Treasury
Seattle                             1700 Corporate Tax
Seattle                             1700 Control And Credit
Seattle                             1700 Shareholder Services
Seattle                             1700 Benefits
Seattle                             1700 Manufacturing
Seattle                             1700 Construction
Seattle                             1700 Contracting
Seattle                             1700 Operations
Seattle                             1700 IT Support
Seattle                             1700 NOC
Seattle                             1700 IT Helpdesk
Seattle                             1700 Government Sales
Seattle                             1700 Retail Sales
Seattle                             1700 Recruiting
Seattle                             1700 Payroll
Toronto                             1800 Marketing
London                              2400 Human Resources
Oxford                              2500 Sales
Munich                              2700 Public Relations
```

2. Joining the `locations` and `countries` tables, display the country name and city.

```
COUNTRY_NAME              CITY
------------------------- ---------------------
Australia                 Sydney
Brazil                    Sao Paulo
Canada                    Toronto
Canada                    Whitehorse
Switzerland               Geneva
Switzerland               Bern
China                     Beijing
Germany                   Munich
India                     Bombay
Italy                     Roma
Italy                     Venice
Japan                     Tokyo
Japan                     Hiroshima
Mexico                    Mexico City
Netherlands               Utrecht
Singapore                 Singapore
United Kingdom            London
United Kingdom            Oxford
United Kingdom            Stretford
United States of America  Southlake
United States of America  South San Francisco
United States of America  South Brunswick
United States of America  Seattle
```

3. Joining the `locations`, `countries`, and `departments` tables, display the country name, city, and department name.

```
COUNTRY_NAME              CITY                  DEPARTMENT_NAME
------------------------- --------------------- --------------------
United States of America  Southlake             IT
United States of America  South San Francisco   Shipping
United States of America  Seattle               Administration
United States of America  Seattle               Purchasing
United States of America  Seattle               Executive
United States of America  Seattle               Finance
United States of America  Seattle               Accounting
United States of America  Seattle               Treasury
United States of America  Seattle               Corporate Tax
United States of America  Seattle               Control And Credit
United States of America  Seattle               Shareholder Services
United States of America  Seattle               Benefits
United States of America  Seattle               Manufacturing
United States of America  Seattle               Construction
United States of America  Seattle               Contracting
United States of America  Seattle               Operations
United States of America  Seattle               IT Support
United States of America  Seattle               NOC
United States of America  Seattle               IT Helpdesk
United States of America  Seattle               Government Sales
United States of America  Seattle               Retail Sales
United States of America  Seattle               Recruiting
United States of America  Seattle               Payroll
Canada                    Toronto               Marketing
United Kingdom            London                Human Resources
United Kingdom            Oxford                Sales
Germany                   Munich                Public Relations
```

4. Joining the `employees` and `job_history` tables, display the employee ID, first and last name, and the job ID. Display the output in sequence by employee_id.

```
EMPLOYEE_ID FIRST_NAME           LAST_NAME            JOB_ID
----------- -------------------- -------------------- ----------
        101 Neena                Kochhar              AC_MGR
        101 Neena                Kochhar              AC_ACCOUNT
        102 Lex                  De Haan              IT_PROG
        114 Den                  Raphaely             ST_CLERK
        122 Payam                Kaufling             ST_CLERK
        176 Jonathon             Taylor               SA_REP
        176 Jonathon             Taylor               SA_MAN
        200 Jennifer             Whalen               AD_ASST
        200 Jennifer             Whalen               AC_ACCOUNT
        201 Michael              Hartstein            MK_REP
```

5. Joining the `jobs` and `job_history` tables, display the job title, employee ID, and starting date for all employees who started in that job after Jan. 1, 1998.

```
JOB_TITLE                           EMPLOYEE_ID START_DATE
----------------------------------- ----------- -----------
Public Accountant                           200 01-JUL-2002
Accounting Manager                          101 28-OCT-2001
Programmer                                  102 13-JAN-2001
Marketing Representative                    201 17-FEB-2004
Sales Manager                               176 01-JAN-2007
Sales Representative                        176 24-MAR-2006
Stock Clerk                                 114 24-MAR-2006
Stock Clerk                                 122 01-JAN-2007
```

6. Modify the above query: remove the start date restriction and also include the employees’ first and last names.

```
JOB_TITLE                 EMPLOYEE_ID START_DATE  FIRST_NAME    LAST_NAME
------------------------- ----------- ----------- ------------- -------------
Accounting Manager                101 28-OCT-2001 Neena         Kochhar
Public Accountant                 101 21-SEP-1997 Neena         Kochhar
Programmer                        102 13-JAN-2001 Lex           De Haan
Stock Clerk                       114 24-MAR-2006 Den           Raphaely
Stock Clerk                       122 01-JAN-2007 Payam         Kaufling
Sales Representative              176 24-MAR-2006 Jonathon      Taylor
Sales Manager                     176 01-JAN-2007 Jonathon      Taylor
Administration Assistant          200 17-SEP-1995 Jennifer      Whalen
Public Accountant                 200 01-JUL-2002 Jennifer      Whalen
Marketing Representative          201 17-FEB-2004 Michael       Hartstein
```

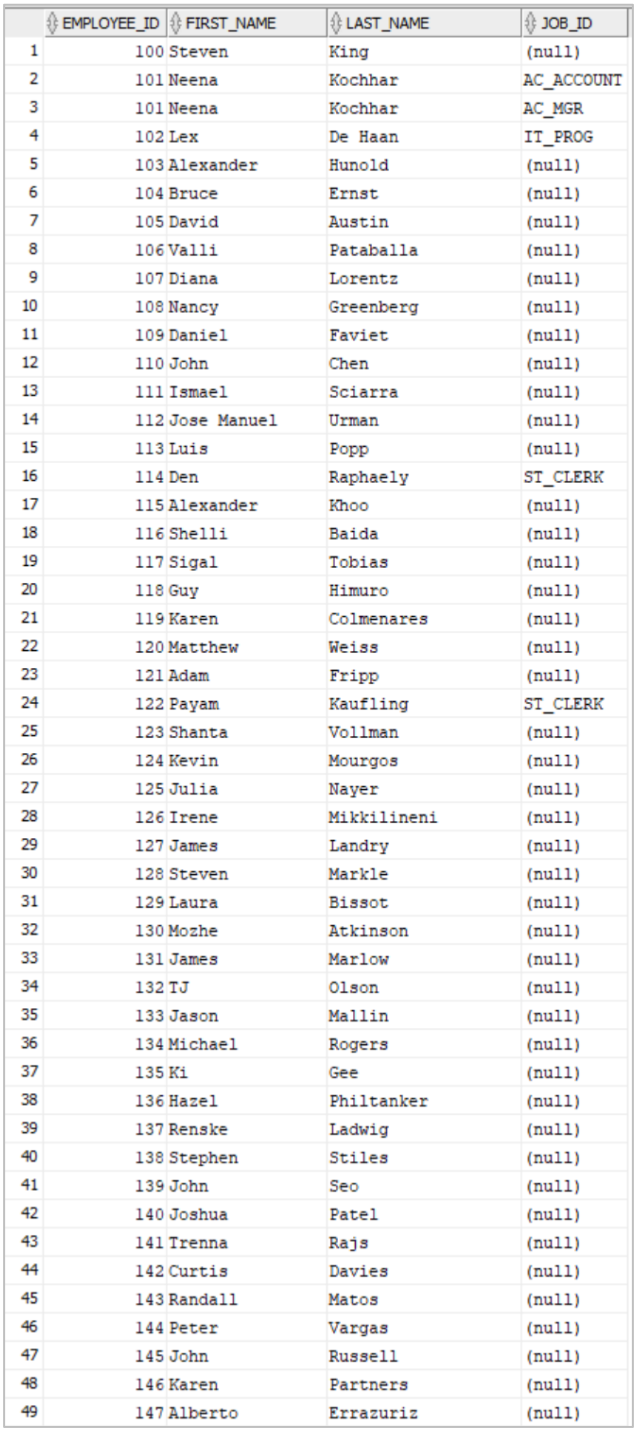
Congratulations! You have completed this exercise.

Exercise 4.2: Using `OUTER JOINs`

Connect to the `HR` account.

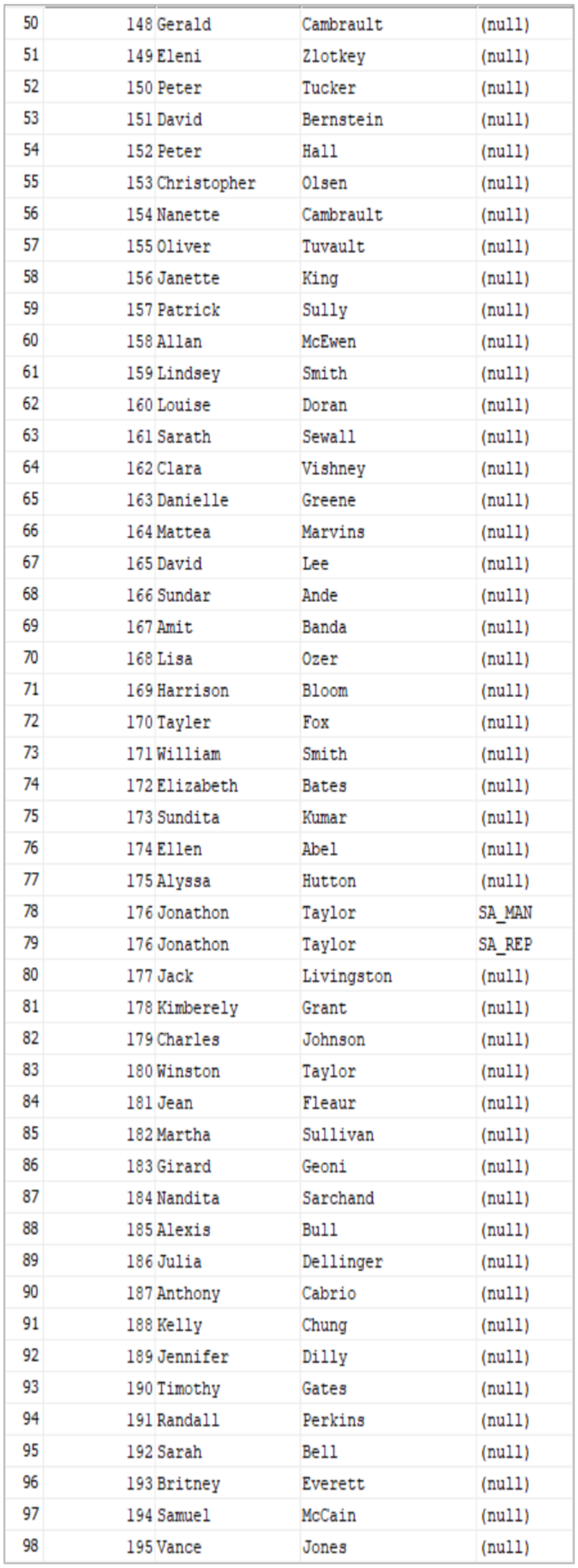
Using the standard outer join syntax, write queries to display the following information:

1. Joining the `employees` and `job_history` tables, display the employee ID, first and last name, and the job ID. Include all employees, whether or not they have any job history. Display in employee ID order. Modify your solution to Step 4 of the previous exercise.

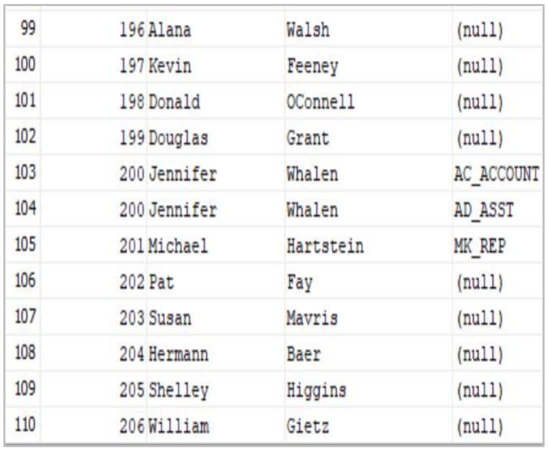


	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID
1	100	Steven	King	(null)
2	101	Neena	Kochhar	AC_ACCOUNT
3	101	Neena	Kochhar	AC_MGR
4	102	Lex	De Haan	IT_PROG
5	103	Alexander	Hunold	(null)
6	104	Bruce	Ernst	(null)
7	105	David	Austin	(null)
8	106	Valli	Pataballa	(null)
9	107	Diana	Lorentz	(null)
10	108	Nancy	Greenberg	(null)
11	109	Daniel	Faviet	(null)
12	110	John	Chen	(null)
13	111	Ismael	Sciarra	(null)
14	112	Jose Manuel	Urman	(null)
15	113	Luis	Popp	(null)
16	114	Den	Raphaely	ST_CLERK
17	115	Alexander	Khoo	(null)
18	116	Shelli	Baida	(null)
19	117	Sigal	Tobias	(null)
20	118	Guy	Himuro	(null)
21	119	Karen	Colmenares	(null)
22	120	Matthew	Weiss	(null)
23	121	Adam	Fripp	(null)
24	122	Payam	Kaufling	ST_CLERK
25	123	Shanta	Vollman	(null)
26	124	Kevin	Mourgos	(null)
27	125	Julia	Nayer	(null)
28	126	Irene	Mikkilineni	(null)
29	127	James	Landry	(null)
30	128	Steven	Markle	(null)
31	129	Laura	Bissot	(null)
32	130	Mozhe	Atkinson	(null)
33	131	James	Marlow	(null)
34	132	TJ	Olson	(null)
35	133	Jason	Mallin	(null)
36	134	Michael	Rogers	(null)
37	135	Ki	Gee	(null)
38	136	Hazel	Philtanker	(null)
39	137	Renske	Ladwig	(null)
40	138	Stephen	Stiles	(null)
41	139	John	Seo	(null)
42	140	Joshua	Patel	(null)
43	141	Trenna	Rajs	(null)
44	142	Curtis	Davies	(null)
45	143	Randall	Matos	(null)
46	144	Peter	Vargas	(null)
47	145	John	Russell	(null)
48	146	Karen	Partners	(null)
49	147	Alberto	Errazuriz	(null)

Continued on the next page



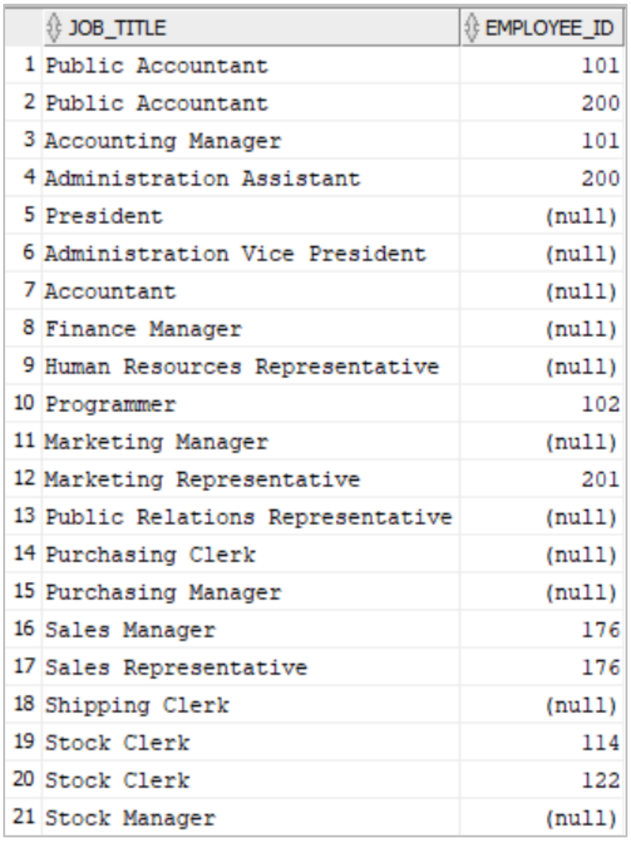
50	148	Gerald	Cambrault	(null)
51	149	Eleni	Zlotkey	(null)
52	150	Peter	Tucker	(null)
53	151	David	Bernstein	(null)
54	152	Peter	Hall	(null)
55	153	Christopher	Olsen	(null)
56	154	Nanette	Cambrault	(null)
57	155	Oliver	Tuvault	(null)
58	156	Janette	King	(null)
59	157	Patrick	Sully	(null)
60	158	Allan	McEwen	(null)
61	159	Lindsey	Smith	(null)
62	160	Louise	Doran	(null)
63	161	Sarath	Sewall	(null)
64	162	Clara	Vishney	(null)
65	163	Danielle	Greene	(null)
66	164	Mattea	Marvins	(null)
67	165	David	Lee	(null)
68	166	Sundar	Ande	(null)
69	167	Amit	Banda	(null)
70	168	Lisa	Ozer	(null)
71	169	Harrison	Bloom	(null)
72	170	Tayler	Fox	(null)
73	171	William	Smith	(null)
74	172	Elizabeth	Bates	(null)
75	173	Sundita	Kumar	(null)
76	174	Ellen	Abel	(null)
77	175	Alyssa	Hutton	(null)
78	176	Jonathon	Taylor	SA_MAN
79	176	Jonathon	Taylor	SA_REP
80	177	Jack	Livingston	(null)
81	178	Kimberely	Grant	(null)
82	179	Charles	Johnson	(null)
83	180	Winston	Taylor	(null)
84	181	Jean	Fleaur	(null)
85	182	Martha	Sullivan	(null)
86	183	Girard	Geoni	(null)
87	184	Nandita	Sarchand	(null)
88	185	Alexis	Bull	(null)
89	186	Julia	Dellinger	(null)
90	187	Anthony	Cabrio	(null)
91	188	Kelly	Chung	(null)
92	189	Jennifer	Dilly	(null)
93	190	Timothy	Gates	(null)
94	191	Randall	Perkins	(null)
95	192	Sarah	Bell	(null)
96	193	Britney	Everett	(null)
97	194	Samuel	McCain	(null)
98	195	Vance	Jones	(null)



99	196	Alana	Walsh	(null)
100	197	Kevin	Feeney	(null)
101	198	Donald	OConnell	(null)
102	199	Douglas	Grant	(null)
103	200	Jennifer	Whalen	AC_ACCOUNT
104	200	Jennifer	Whalen	AD_ASST
105	201	Michael	Hartstein	MK_REP
106	202	Pat	Fay	(null)
107	203	Susan	Mavris	(null)
108	204	Hermann	Baer	(null)
109	205	Shelley	Higgins	(null)
110	206	William	Gietz	(null)

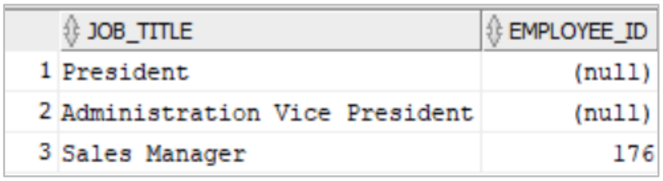
```
110 rows selected.
```

2. Joining the `jobs` and `job_history` tables, display the job title and employee ID for all jobs, whether or not they have job history.



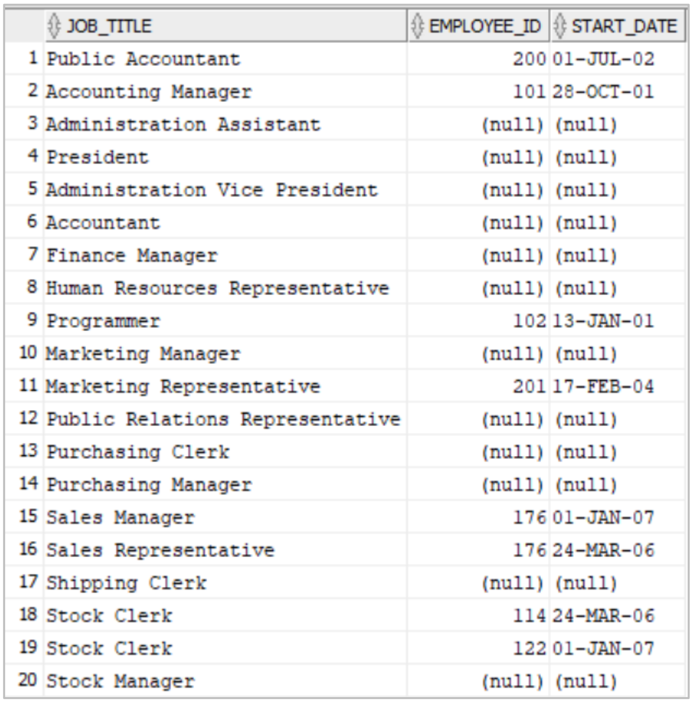
	JOB_TITLE	EMPLOYEE_ID
1	Public Accountant	101
2	Public Accountant	200
3	Accounting Manager	101
4	Administration Assistant	200
5	President	(null)
6	Administration Vice President	(null)
7	Accountant	(null)
8	Finance Manager	(null)
9	Human Resources Representative	(null)
10	Programmer	102
11	Marketing Manager	(null)
12	Marketing Representative	201
13	Public Relations Representative	(null)
14	Purchasing Clerk	(null)
15	Purchasing Manager	(null)
16	Sales Manager	176
17	Sales Representative	176
18	Shipping Clerk	(null)
19	Stock Clerk	114
20	Stock Clerk	122
21	Stock Manager	(null)

3. Modify the above query to restrict the result set to jobs whose minimum salary exceeds 9000.



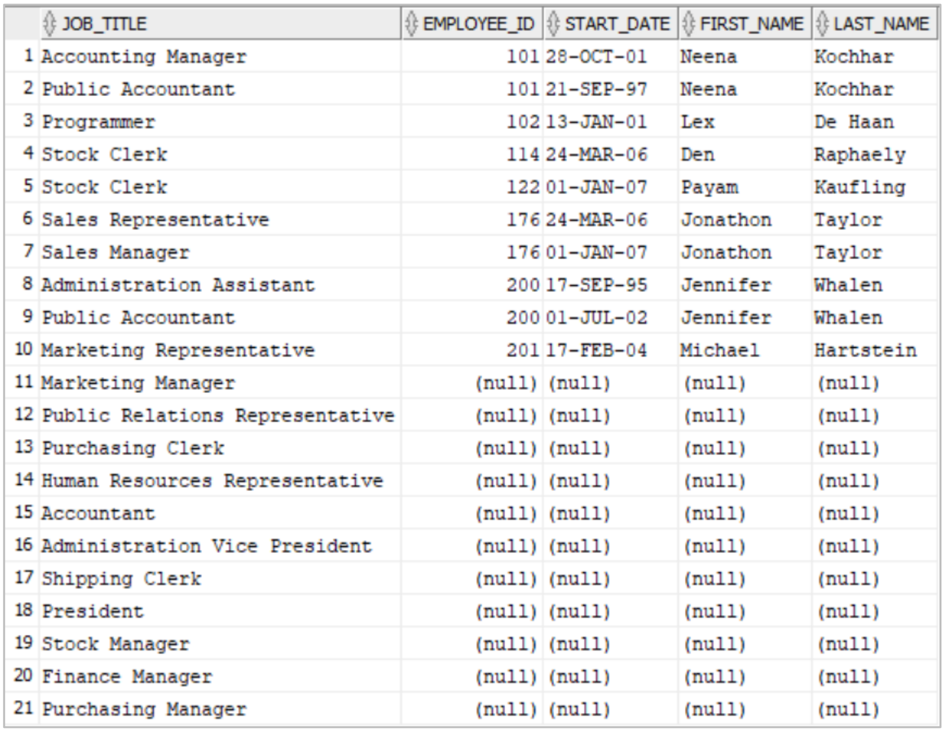
	JOB_TITLE	EMPLOYEE_ID
1	President	(null)
2	Administration Vice President	(null)
3	Sales Manager	176

4. Joining the `jobs` and `job_history` tables, display the job title, employee ID, and starting date for all employees who started in that job after Jan. 1, 1998. Include jobs ***even if*** they do not have any history.



	JOB_TITLE	EMPLOYEE_ID	START_DATE
1	Public Accountant	200	01-JUL-02
2	Accounting Manager	101	28-OCT-01
3	Administration Assistant	(null)	(null)
4	President	(null)	(null)
5	Administration Vice President	(null)	(null)
6	Accountant	(null)	(null)
7	Finance Manager	(null)	(null)
8	Human Resources Representative	(null)	(null)
9	Programmer	102	13-JAN-01
10	Marketing Manager	(null)	(null)
11	Marketing Representative	201	17-FEB-04
12	Public Relations Representative	(null)	(null)
13	Purchasing Clerk	(null)	(null)
14	Purchasing Manager	(null)	(null)
15	Sales Manager	176	01-JAN-07
16	Sales Representative	176	24-MAR-06
17	Shipping Clerk	(null)	(null)
18	Stock Clerk	114	24-MAR-06
19	Stock Clerk	122	01-JAN-07
20	Stock Manager	(null)	(null)

5. Modify the above query: remove the start date restriction and also include the employee's first and last names.

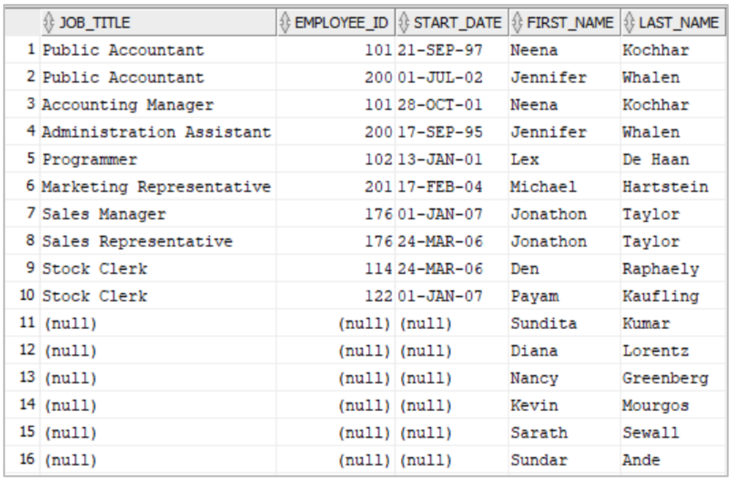


	JOB_TITLE	EMPLOYEE_ID	START_DATE	FIRST_NAME	LAST_NAME
1	Accounting Manager	101	28-OCT-01	Neena	Kochhar
2	Public Accountant	101	21-SEP-97	Neena	Kochhar
3	Programmer	102	13-JAN-01	Lex	De Haan
4	Stock Clerk	114	24-MAR-06	Den	Raphaely
5	Stock Clerk	122	01-JAN-07	Payam	Kaufling
6	Sales Representative	176	24-MAR-06	Jonathon	Taylor
7	Sales Manager	176	01-JAN-07	Jonathon	Taylor
8	Administration Assistant	200	17-SEP-95	Jennifer	Whalen
9	Public Accountant	200	01-JUL-02	Jennifer	Whalen
10	Marketing Representative	201	17-FEB-04	Michael	Hartstein
11	Marketing Manager	(null)	(null)	(null)	(null)
12	Public Relations Representative	(null)	(null)	(null)	(null)
13	Purchasing Clerk	(null)	(null)	(null)	(null)
14	Human Resources Representative	(null)	(null)	(null)	(null)
15	Accountant	(null)	(null)	(null)	(null)
16	Administration Vice President	(null)	(null)	(null)	(null)
17	Shipping Clerk	(null)	(null)	(null)	(null)
18	President	(null)	(null)	(null)	(null)
19	Stock Manager	(null)	(null)	(null)	(null)
20	Finance Manager	(null)	(null)	(null)	(null)
21	Purchasing Manager	(null)	(null)	(null)	(null)

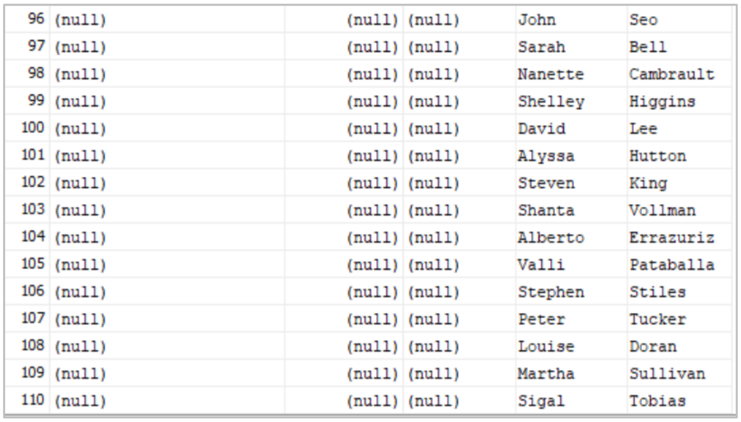
Bonus Exercise (if time permits)

6. Joining the `employees`, `job_history`, and `jobs` tables, display the job title, employee ID, start date, and employee first and last names for ALL employees, whether or not they have any job history.

Hint: *You will need to change the most important table.*



	JOB_TITLE	EMPLOYEE_ID	START_DATE	FIRST_NAME	LAST_NAME
1	Public Accountant	101	21-SEP-97	Neena	Kochhar
2	Public Accountant	200	01-JUL-02	Jennifer	Whalen
3	Accounting Manager	101	28-OCT-01	Neena	Kochhar
4	Administration Assistant	200	17-SEP-95	Jennifer	Whalen
5	Programmer	102	13-JAN-01	Lex	De Haan
6	Marketing Representative	201	17-FEB-04	Michael	Hartstein
7	Sales Manager	176	01-JAN-07	Jonathon	Taylor
8	Sales Representative	176	24-MAR-06	Jonathon	Taylor
9	Stock Clerk	114	24-MAR-06	Den	Raphaely
10	Stock Clerk	122	01-JAN-07	Payam	Kaufling
11	(null)	(null)	(null)	Sundita	Kumar
12	(null)	(null)	(null)	Diana	Lorentz
13	(null)	(null)	(null)	Nancy	Greenberg
14	(null)	(null)	(null)	Kevin	Mourgos
15	(null)	(null)	(null)	Sarath	Sewall
16	(null)	(null)	(null)	Sundar	Ande



96	(null)	(null)	(null)	John	Seo
97	(null)	(null)	(null)	Sarah	Bell
98	(null)	(null)	(null)	Nanette	Cambrault
99	(null)	(null)	(null)	Shelley	Higgins
100	(null)	(null)	(null)	David	Lee
101	(null)	(null)	(null)	Alyssa	Hutton
102	(null)	(null)	(null)	Steven	King
103	(null)	(null)	(null)	Shanta	Vollman
104	(null)	(null)	(null)	Alberto	Errazuriz
105	(null)	(null)	(null)	Valli	Pataballa
106	(null)	(null)	(null)	Stephen	Stiles
107	(null)	(null)	(null)	Peter	Tucker
108	(null)	(null)	(null)	Louise	Doran
109	(null)	(null)	(null)	Martha	Sullivan
110	(null)	(null)	(null)	Sigal	Tobias

```
110 rows selected.
```

Congratulations! You have completed this exercise.

Chapter 5: Additional SQL Functions

Exercise 5.1: Additional SQL Functions

Connect to the `HR` account.

1. Write a query to display the department ID, first and last, names, hire date, and commission percentage of all employees with manager 100 (Steven King) whose hire date is within 2 years of Jan 1, 2007. Sequence the list in department, hire date order. Use a non-standard function to display null commissions as 0.

```
DEPARTMENT_ID FIRST_NAME  LAST_NAME  HIRE_DATE   COMMISSION
------------- ----------- ---------- ----------- ----------
           50 Adam        Fripp      10-APR-2005          0
           50 Shanta      Vollman    10-OCT-2005          0
           50 Kevin       Mourgos    16-NOV-2007          0
           80 Karen       Partners   05-JAN-2005         .3
           80 Alberto     Errazuriz  10-MAR-2005         .3
           80 Gerald      Cambrault  15-OCT-2007         .3
           80 Eleni       Zlotkey    29-JAN-2008         .2
           90 Neena       Kochhar    21-SEP-2005          0
```

2. Change the previous query to use a standard function to display the commission percentage. This time, order the list so that those hired closest to Jan 1, 2007, are listed first.

```
DEPARTMENT_ID FIRST_NAME  LAST_NAME  HIRE_DATE   COMMISSION
------------- ----------- ---------- ----------- ----------
           80 Gerald      Cambrault  15-OCT-2007         .3
           50 Kevin       Mourgos    16-NOV-2007          0
           80 Eleni       Zlotkey    29-JAN-2008         .2
           50 Shanta      Vollman    10-OCT-2005          0
           90 Neena       Kochhar    21-SEP-2005          0
           50 Adam        Fripp      10-APR-2005          0
           80 Alberto     Errazuriz  10-MAR-2005         .3
           80 Karen       Partners   05-JAN-2005         .3
```

Congratulations! You have completed this exercise.

This page intentionally left blank.

Chapter 6: Data Manipulation Language

Exercise 6.1: Manipulating Data

Connect to the `HR` account.

1. Display all the rows in the `regions` table.
2. Add a new row for Central America. Make it ID 5.
3. Display all the rows in the `regions` table.
4. Add a new row for South America. Make it ID 6.
5. Display all the rows in the `regions` table.
6. Update all `regions` rows with the name, Central America. Change their name to South and Central America.
7. Display all the rows in the `regions` table.
8. Delete the `regions` row whose ID is 6.
9. Display all the rows in the `regions` table.
10. Issue a ROLLBACK and re-display the `regions` table.

Congratulations! You have completed this exercise.

This page intentionally left blank.

Chapter 7: Databases with JDBC (Java Database Connectivity)

Exercise 7.1: Connecting to a Database

In this exercise, you will write the Java code to connect to the database.

Over the next two days, you will enhance this project to use and test a Data Access Object (DAO), a `TransactionManager`, and a Business Service.

This exercise will complete the SimpleDataSource class that provides a database Connection for the DAO to use. In a real-world project, you will not be creating a DataSource yourself. In most situations, an application server will create a DataSource and make it available for application programmers to use.

For the most part, the details of our SimpleDataSource are not that important. The SimpleDataSource is not part of the actual application source code. Instead, it is in the testing section of the Eclipse project because it will only be used during testing.

To make the SimpleDataSource functional, you will implement the `openConnection()` method. The instructions below will show you how to do this in a Test-Driven Development (TDD) style.

1. Open the EmployeeManagement project in Eclipse.
   a. Verify that the Oracle dependency is listed in `pom.xml`.

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>18.3.0.0</version>
</dependency>
```

2. Open the `SimpleDataSourceTest.java` file.
   a. It is located in the `com.fidelity.integration` package in `src/test/java`.

3. Verify the `@AfterEach` method calls the `shutdown` method on the `SimpleDataSource` object.
   a. This ensures that the database `Connection` is closed after each test.

4. Run the test in this file.
   a. You should get a **Green** bar.
   b. This shows that the `SimpleDataSource` object is successfully created.

5. Open the `SimpleDataSource.java` file.
    a. In the `com.fidelity.integration` package in `src/test/java`.

6. Examine the `openConnection()` method.
    a. You will notice that the method is not entirely functional at this point.
    b. The `openConnection` method is private.
    c. The `getConnection` method calls the `openConnection` method.

7. In `SimpleDataSourceTest.java`, write a JUnit test to get a `Connection` from the `SimpleDataSource`.
    a. By calling the `getConnection` method.
    b. Assert that the `Connection` that is returned is not null.
    c. Run the test.
    d. Naturally, the test will fail!
    e. But of course, you expected that.
    f. **Red** bar, …

8. Complete the `openConnection` method to return a `Connection` to the Oracle database.
    a. Review the file `db.properties`.
        i. This file is in the `src/main/resources` folder.
        ii. It will be in the classpath at runtime.
    b. Call `DriverManager.getConnection` to create the connection.
        iii. Start with the simplest code that could possibly work. See the example in the course notes that defines the connection parameters as hard-coded strings.
        iv. After you get the **Green** bar for your test case, refactor your `getConnection` method to read the connection parameters from a properties file. See the second example in the course notes.
    c. If an exception occurs, throw a `DatabaseException`.

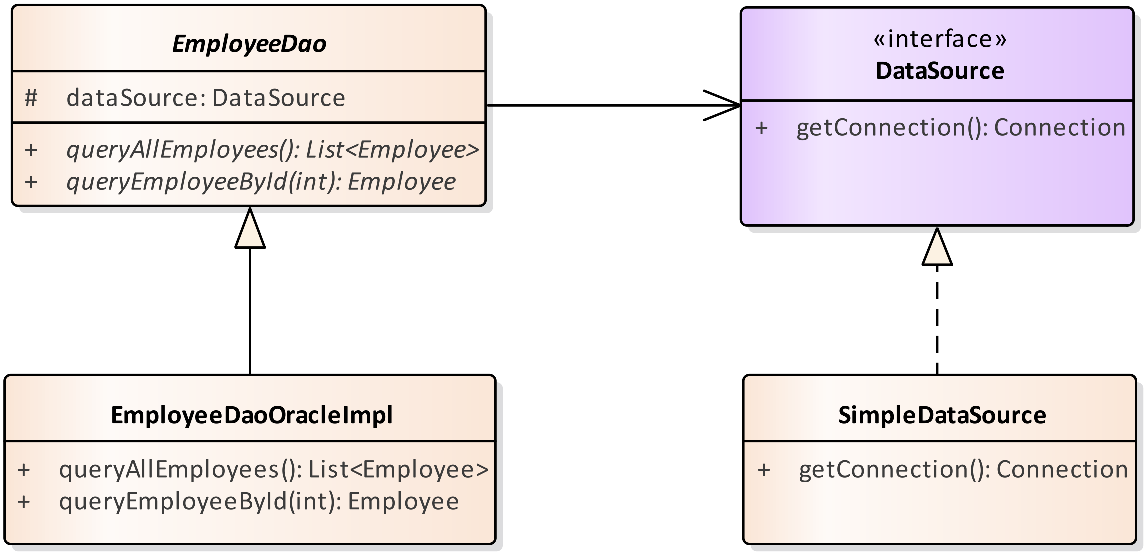
9. Run your test again.
    a. Of course, you now get a **Green** bar because you wrote perfect code!
    b. **Red** bar, **Green** bar, …

Congratulations! You have completed this exercise.

Exercise 7.2: Creating Objects from a Database Query

In this exercise, you will perform queries to access employee data from the scott.emp table.

The UML class diagram for this exercise is shown below:


EmployeeDao
# dataSource: DataSource
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
«interface»
DataSource
+ getConnection(): Connection
EmployeeDaoOracleImpl
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
SimpleDataSource
+ getConnection(): Connection

1. Open the SQL Developer application.
   a. View the structure of the emp table in the scott database.
   b. In a worksheet, write a query that will select all of the records in the emp table.
      i. Do **not** use SELECT *.
      ii. List the columns by name in the SELECT statement.

2. Continue working with the EmployeeManagement project in Eclipse.

3. Examine the Employee class.
   a. This is the Java representation of the scott.emp table.

4. Examine the EmployeeDao.java file.
   a. Note that the class is abstract.
   b. The two query methods are also abstract.
   c. There is a protected DataSource field.

5. Create a new Java class in the `com.fidelity.integration` package to hold the database access code.
   a. Name it `EmployeeDaoOracleImpl`
   b. Have it extend the `EmployeeDao` base class.
   c. Put it in the `com.fidelity.integration` package.
      i. In all the solution projects, we will use `com.fidelity.integration` to show that it is our integration layer.
   d. Let Eclipse write the two abstract methods inherited from the base class.

6. Now, create a test class for the `EmployeeDaoOracleImpl`.
   a. Right-click `EmployeeDaoOracleImpl` and select **New | Other…**, then navigate to **Java | Junit** and select **Junit Test Case**.
   b. Make sure you are creating a JUnit 5 (Jupiter) test case.
   c. Check that the name of the class is `EmployeeDaoQueryTest`, and it is in the same package as `EmployeeDaoOracleImpl`.
   d. Change the Source folder to `EmployeeManagement/src/test/java`.

7. Check the `@BeforeEach` and `@AfterEach` checkboxes.

8. In the `@BeforeEach` method, create a `SimpleDataSource` and an `EmployeeDaoOracleImpl` object.

9. In the `@AfterEach` methods, call the shutdown() method on the `SimpleDataSource` object.

10. Write a Test method to retrieve all records from the `scott.emp` table and return a list of `Employee` objects.
    a. Work in TDD fashion.
    b. How will you tell whether the method returned the right set of employees?

11. Run the test.
    a. You got the **Red** bar, didn't you?

12. Write the implementation of the method in `EmployeeDaoOracleImpl`.
    a. Use the sample code in the course notes as a guide for writing this new method.
    b. Use the query that you created in SQL Developer.
       i. Remember to delete the trailing **"`;`"**
    c. Of course, you will use a PreparedStatement!

    d. Although they are not the only nullable columns, both `mgr` and `comm` have null values. JDBC will treat these as 0, which is OK for now.

13. Run the test
    a. Got the **Green** bar? Great! Go to Step 14.
    b. Got a **Red** bar?
        i. Read the error message.
        ii. Do some debugging.
        iii. Get that coveted **Green** bar.

14. Create a method to retrieve a single `scott.emp` record by `empno` and return a single `Employee` object.
    a. Again, work in TDD fashion. What tests do you need?
    b. Write the query in SQL Developer first, then copy into the DAO method.
        i. If a SQL statement does not work in SQL Developer, it will definitely not work in your JDBC code!
    c. Write the tests, and then let Eclipse create the method for you.

15. Work until you get that **Green** bar.

Optional steps, if you have time:

16. Create a method to retrieve all the `scott.emp` records where the employee name matches a string that is passed in as a parameter. Return a list of `Employee` objects.
    a. Even though all the names are actually unique, this is not guaranteed by the database, so you should return a list, not a single `Employee` object.

17. Add a test that proves that SQL injection is not possible. Consider the string `"BobbyTables' OR '1' = '1"`.

18. Create a method to retrieve all the `scott.emp` records in a particular department and return a list of `Employee` objects.

Congratulations! You have completed this exercise.

This page was intentionally left blank.

Chapter 8: Updating Databases

Exercise 8.1: Inserting a Record with JDBC

Since the `insertEmployee` method will insert a new record into the database, for testing, we should run the test in a transaction and roll back the transaction after the test has completed.

If you make significant changes to the `scott.emp` table, you can use the file `sql\emp_populate.sql` to recover the table.

In this exercise, you will use a `TransactionManager` to start and roll back a transaction.

The TransactionManager class lives up to its name by starting, committing, and rolling back transactions.

The DAO class will not call on (or even know of) the `TransactionManager`.

1. Create a new class `TransactionManager` in the `com.fidelity.integration` package.

2. Define a constructor for `TransactionManager` that has a `DataSource` argument and initializes a data field.

3. Define the `startTransaction`, `commitTransaction`, and `rollbackTransacton` methods.
   a. Use the `Connection` from the `DataSource` in the transaction methods.

4. Open the `EmployeeDaoDMLTest.java` file.

5. In the `@BeforeEach` method, create a new `EmployeeDaoOracleImpl` object and assign it to the DAO data member.

6. Call the `TransactionManager` `startTransaction()` in `@BeforeEach`.

7. Call the `TransactionManager` `rollbackTransaction()` in `@AfterEach`.

8. Add the `insertEmployee` method to the `EmployeeDao` base class.

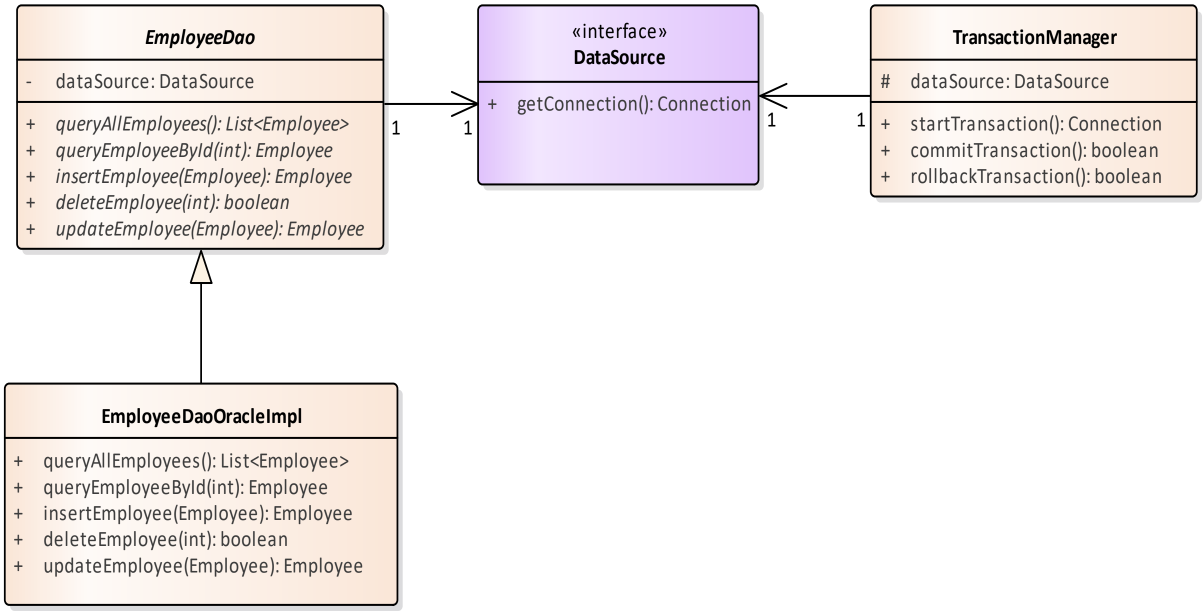
9. Write a test for the `insertEmployee` method in `EmployeeDaoDMLTest`.

10. Let Eclipse write the method in `EmployeeDaoOracleImpl`.

11. Run the test to get the **Red** bar.

12. Define the new method correctly in `EmployeeDaoOracleImpl`.

13. Run the test to get the **Green** bar.

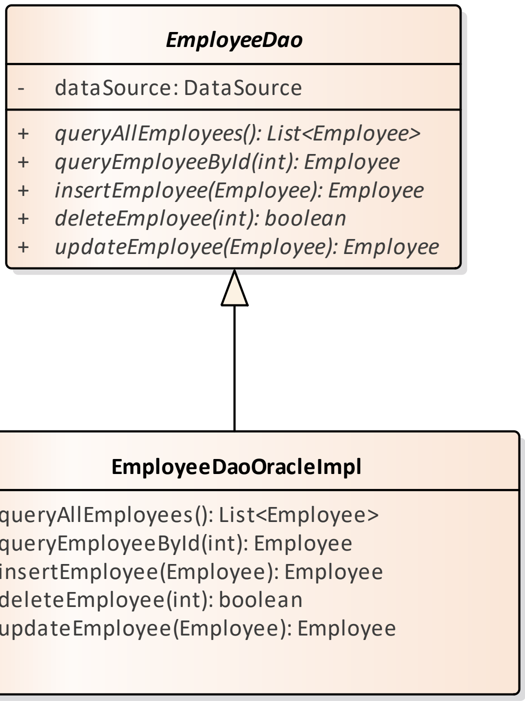

EmployeeDao
- dataSource: DataSource
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
+ insertEmployee(Employee): Employee
+ deleteEmployee(int): boolean
+ updateEmployee(Employee): Employee
1
1
«interface»
DataSource
+ getConnection(): Connection
1
1
TransactionManager
# dataSource: DataSource
+ startTransaction(): Connection
+ commitTransaction(): boolean
+ rollbackTransaction(): boolean
EmployeeDaoOracleImpl
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
+ insertEmployee(Employee): Employee
+ deleteEmployee(int): boolean
+ updateEmployee(Employee): Employee

Congratulations! You have completed this exercise.

Exercise 8.2: Updating and Deleting Records

In this exercise, you will continue your work on the `EmployeeManagement_DML_Test` class and the `EmployeeDaoOracleImpl` class.

1. Add the abstract `updateEmployee` method to the `EmployeeDao` class.
2. Write a test for the new method.
3. Implement the new method in the `EmployeeDaoOracleImpl`.
4. Get that **Green** bar.
5. Add the `deleteEmployee` method to the `EmployeeDao` class.
6. Write a test for the new method.
7. Implement the new method in the `EmployeeDaoOracleImpl`.
8. Get that **Green** bar.
9. Write a negative test to verify the DAO throws a `DatabaseException`.
   a. Attempt to update an employee to be in a non-existent department.
   b. Verify the `DatabaseException` is thrown.


EmployeeDao
- dataSource: DataSource
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
+ insertEmployee(Employee): Employee
+ deleteEmployee(int): boolean
+ updateEmployee(Employee): Employee
EmployeeDaoOracleImpl
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
+ insertEmployee(Employee): Employee
+ deleteEmployee(int): boolean
+ updateEmployee(Employee): Employee

Congratulations! You have completed this exercise.

Exercise 8.3: Working with Multi-Table Queries

In this exercise, you will extract information from two database tables.

In this scenario, a query will extract data from two tables. One record in each table will be involved in the query. A record in the `emp` table will be associated with one record in the `emp_perf` table.

1. Use SQL Developer to open the `emp_perf.sql` file that is in the `sql` folder in the `EmployeeManagement` Eclipse project.
2. Run the `emp_perf.sql` script.
   a. This will create a new table named `emp_perf`.
   b. Each record in the `emp_perf` table stores performance evaluation information for an employee.
3. Use a new worksheet in SQL Developer to create a query that will select all the columns in a record in the `emp` table and the `perf_code` column in the `emp_perf` table for that employee.
   a. Use a `JOIN` to obtain the data from the two tables.
   b. Use a `WHERE` clause to specify the employee number (`empno` column).
4. In Eclipse, create an `enum` with values for an employee performance review. When you process the query's result set, you will need to map the integer review code to a value of your new `enum`.
   a. *Hint:* see the `emp_perf` table for the code values.
   b. *Hint:* refer to the Enumerated Types section in Chapter 7 of the course notes.
5. Modify the Employee class to contain a property to store the performance review value for that employee.
   a. This property can be null if there is no record for that employee in the `emp_perf` table.
   b. Create the getter and setter for the new property.
   c. Update the `equals()`, `hashcode()`, and `toString`() methods.
      i. Ask Eclipse to do this.
6. Using TDD, define and test a new DAO method that queries for an `Employee` by `empno` including the performance property.
   a. Set the performance property for the employee to the value in the `emp_perf` table for that employee.
   b. Handle the situation when the employee does not have an entry in the `emp_perf` table.

Congratulations! You have completed this exercise.

Chapter 9: Working with a Data Access Object

Exercise 9.1: Debugging Data Access Object Methods

Time: 20 minutes
Format: Individual programming exercise

Once again, it's Friday afternoon and you are ready to pack up for a well-deserved weekend after your hard work developing a Data Access Object that utilizes JDBC.

Life is good! The weekend is almost here.

Before your weekend can start, there are a few JUnit tests that have to be examined. There are several tests, but there are two failures and one error.

Since you are now a seasoned Java and JDBC developer and debugger, you are not panicking. You know what you have to do to track the problems down and eliminate them ASAP.

1. Examine the code in the `ProblematicEmployee` project that defines the Data Access Object that is communicating with the `scott` database.

2. Run the JUnit tests for the project.

3. Notice that you get some **Green** bars but also a nasty **Red** bar indicating one error and two failures.

4. Dust off your debugging skills and get to work!
   a. Correct one problem at a time.
   b. You may want to use the Debugger to step through the code in order to isolate the problem.

5. Now that you have all **Green,** you are feeling pretty good.

6. Are you completely sure that your DAO code is correct?

7. Examine the set of DAO tests.

8. Are there any other tests that should be written for this DAO?
   a. If so, you know what to do.
   b. **Red bar**, **Green bar**, **Refactor**.

Congratulations! You have completed this exercise.

Optional Exercise 9.2: Testing a Business Service with a Mock DAO

In this exercise, you will continue working with the `EmployeeManagement` project. You will define the `EmployeeManagementService` class and test with a mock DAO.

Now that you have a working (and tested) DAO, it is time to concentrate on how to use the DAO functionality in an application. Introducing a Business Service is a step in that direction.

The methods in the Business Service will correspond to requirements for the application that is being developed. The Business Service methods will call on the DAO to communicate with the back-end database.

During unit testing of the Business Service, we do not want to actually modify the contents of the database. For this reason, we will use Mockito to create a mock DAO that has the same interface as the `EmployeeDao` but does not actually communicate with the database. Mockito makes it easy to test edge conditions and error paths. You'll add service test cases that verify its behavior for an empty employee list and a DAO exception, as well as a positive test case for a non-empty list.

As usual, work using TDD: write a test case first, run it and watch it fail, then modify the method under test to make the test case pass.

1. The `EmployeeManagementService` is a business service that has a dependency on the `EmployeeDao`.

2. Modify `EmployeeManagementServiceTest` so it creates a mock DAO before each test case.
   a. Refer to the examples in the course notes.

3. Add a new test case that verifies that, if the DAO returns a non-empty list of employees, the `EmployeeManagementService getAllEmployees()` method returns the same list.
   a. After writing the test case, add a `getAllEmployees()` method to `EmployeeManagementService`.
   b. Confirm the new unit test now passes.

4. Add a unit test that verifies that, if the DAO `getAllEmployees()` method returns an empty list of employees, the `EmployeeManagementService getAllEmployees()` method throws an `IllegalStateException`. Run the test case, watch it fail, then add the `EmployeeManagementService getAllEmployees()` method to make the test case pass.

5. Add a unit test that verifies that, if the DAO `getAllEmployees()` method throws a `DatabaseException`, the `getAllEmployees()` method throws the same exception. Run the test case, watch it fail, then modify `getAllEmployees()` to make the test case pass.

Bonus Exercise (if time permits)

6. A best practice is not to "leak" details about a class's implementation to its clients. Currently, if the DAO throws a `DatabaseException`, `EmployeeManagementService` passes the same exception back to its client, thus leaking details about its implementation (i.e., the fact that the service uses a database instead of a web service call). Now you'll fix this deficiency.
   a. Define a new `ServiceException` class, which `extends RuntimeException`.
   b. Add a unit test that verifies that, if the DAO `getAllEmployees()` method throws an exception, the `EmployeeManagementService getAllEmployees()` method throws the new `ServiceException`. Run the test case and watch it fail.
   c. Modify `EmployeeManagementService getAllEmployees()` to make the test case pass.

Congratulations! You have completed this exercise.

Exercise 9.3 Putting It All Together

In this exercise, you will create a new project that will contain a Data Access Object that will define methods for queries, inserts, updates, and deletes on an Oracle database.

You will define tests for all of the DAO methods. Of course, you will do this TDD style!

The tests for the DML operations will be performed in a transaction that is rolled back after the test completes.

Your instructor will provide instructions on how to import the starter project into Eclipse.

Congratulations! You have completed this exercise.

Chapter 10: Advanced JDBC

Optional Exercise 10.1: JDBC Code Review

You have been asked to. participate in a code review of a Data Access Object that will be used in airline flight search software. The software will run in every airport check-in kiosk for a major airline.

Open the `FlightTracker` project in Eclipse.

Work in small groups to examine the code in the `FlightSearch.java` source code file.

Feel free to run the JUnit tests that are defined in the `FlightSearchTest.java` file to verify the small set of unit tests pass and produce the coveted **Green** bar.

A spokesperson for your group should be prepared to discuss the group's evaluation of the `FlightSearch` code.

In particular, your evaluation should include any positive aspects of the code (assuming there are some), and any negative aspects of the code (should there be any).

Your group should decide if you would give the `FlightSearch` code the go-ahead for inclusion in the production software installed in the airport kiosks.

If you are not giving approval of the code, then be specific as to why you would not approve of the code and what must be corrected to gain approval.

Congratulations! You have completed this exercise.

Optional Exercise 10.2: One-to-Many Queries

In this exercise, you will once again extract data from two tables. There are two scenarios described below. In both scenarios, one Department object will have a collection of Employee objects.

One of the main challenges in this exercise is to determine how to properly process the ResultSet that is returned from the query.

Scenario 1: One-to-many (1-1..*)
In this scenario, a query will extract data from two tables. One record in the `dept` table will be associated with one to several records in the `emp` table.

1. Use SQL Developer to inspect the structure of the `dept` table in the `scott` database.
2. In your Eclipse project, create a `Department` class to store the data from a record in the `dept` table.
3. In the `Department` class, add a data field to store the `Employees` in that `Department`.
4. Using TDD, write and test a DAO method to query for a `Department` by id.
   a. Return a `Department` object with the collection of `Employees` for that `Department`.
   b. Use SQL Developer to help you write the query.

Scenario 2: One-to-many (1-0..*)
In this scenario, a query will extract data from two tables. One record in the `dept` table will be associated with zero to several records in the `emp` table. All of the records in the `dept` table are to be returned in this scenario, even if a `dept` record is not associated with any `emp` records.

5. Using TDD, write and test a DAO method that will query for all `Departments`.
   a. Return a collection of `Department` objects with `Employees`.
   b. Include `Departments` that do not contain any `Employees`.
   c. Use SQL Developer to help you write the query.

Congratulations! You have completed this exercise.

Optional Exercise 10.3: Testing a Service Proxy with Mockito

The problem is to guarantee that the DML operations provided by the DAO are called in a transaction.

While the `EmployeeManagementService` could call the `startTransaction` and `commitTransaction` methods to ensure the DML operations are performed in a transaction, that seems to leak some implementation details from the DAO into the service.

An alternative is to use a proxy for the service. This gives us the opportunity to discuss design patterns again and to demonstrate the use of one in a “real” situation.

We want the proxy to call `startTransaction` on the `TransactionManager`, then call the business service methods that are to be part of the transaction, then call `commitTransaction` on the `TransactionManager`.

The `insertNewManager()` method in the service class inserts a new Manager and updates the Employees to be managed by the new Manager.

This method definitely requires a transaction to guarantee that all operations complete successfully, or all are rolled back.

If a `DatabaseException` is thrown, the proxy can let it pass on through to the client.

1. Rename the `EmployeeManagementService` class to be `EmployeeManagementServiceImpl`.
   a. Use the **Refactor | Rename** support in Eclipse.

2. Run the tests to verify that all the tests pass.
   a. That is what should happen when you apply Refactoring.

3. Extract the `EmployeeManagementService` interface from `EmployeeManagementServiceImpl`.
   a. Use the **Refactor | Rename** support in Eclipse.

4. Run the tests to see that **Green** bar!

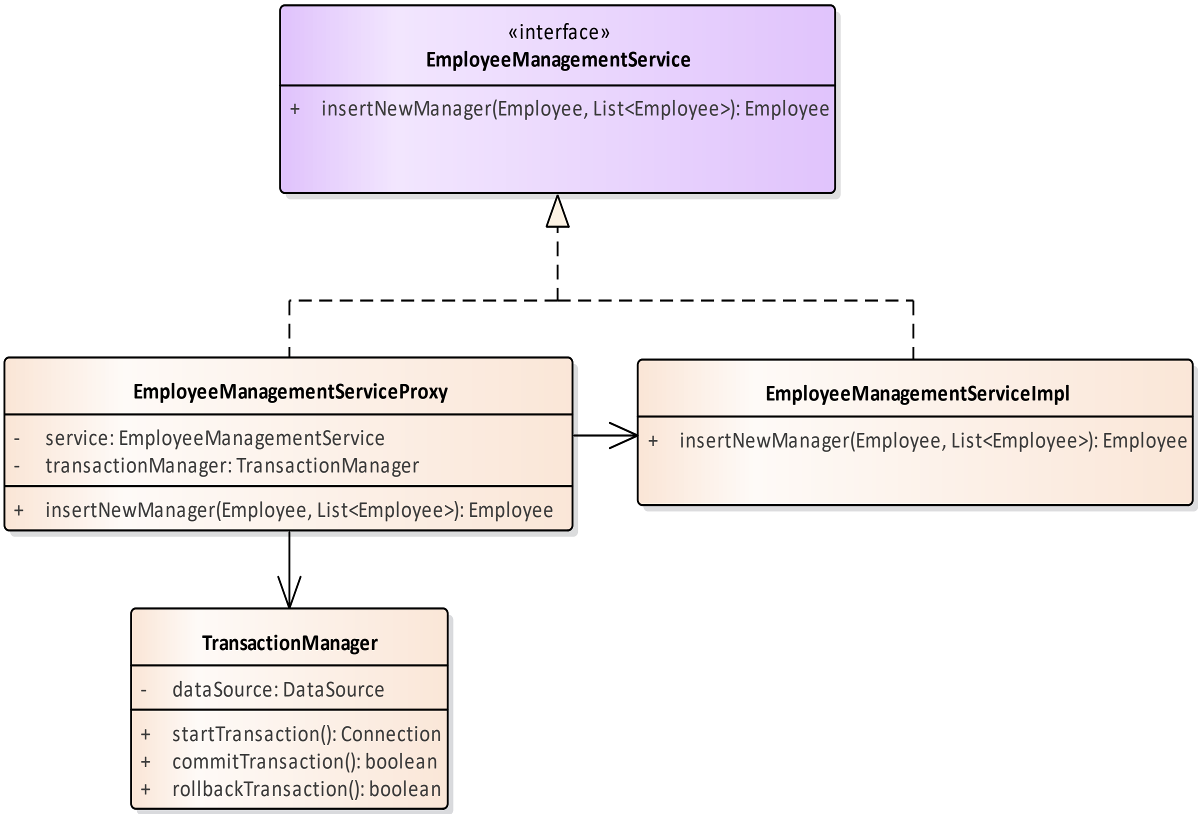
5. Create a new class named `EmployeeManagementServiceProxy`.
   d. The `EmployeeManagementServiceProxy` class implements the `EmployeeManagementService` interface.
   e. The `EmployeeManagementServiceProxy` class has an association with `EmployeeManagementServiceImpl`.

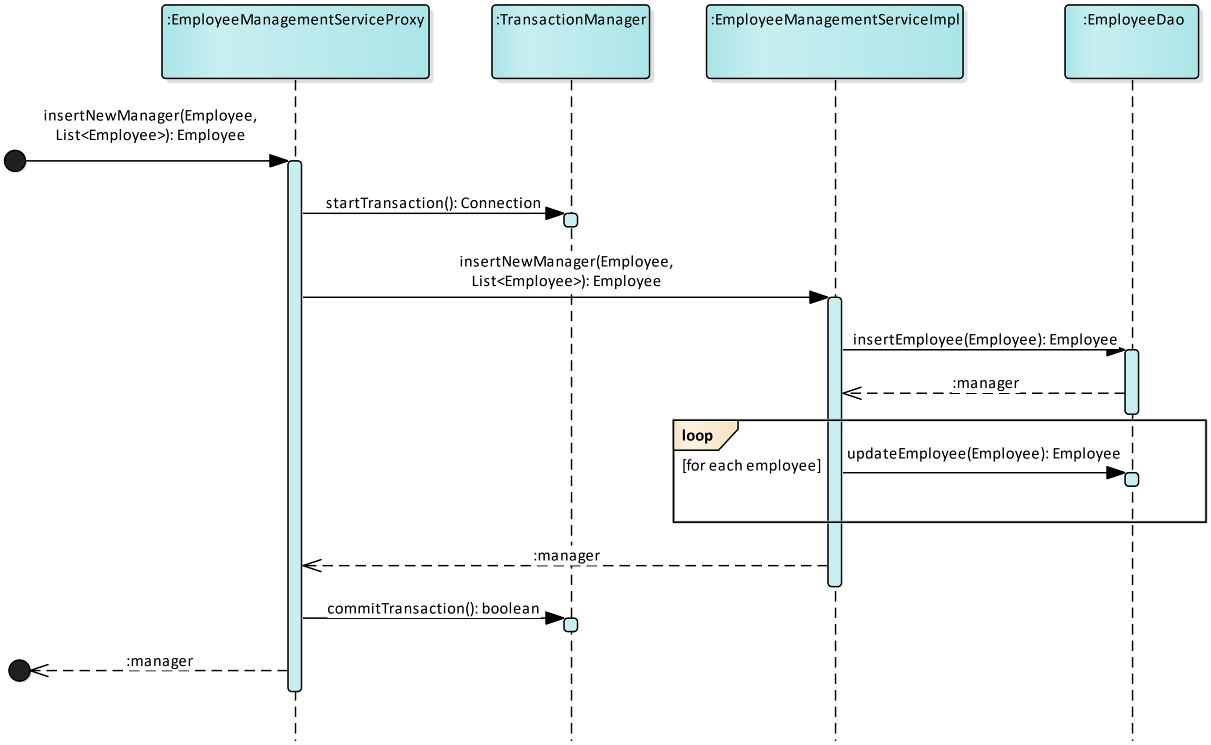
6. Open the `EmployeeManagementServiceProxyTest.java` file.

7. Uncomment the lines of code that:
   a. Add a data field for an `EmployeeManagementServiceProxy`.
   b. Add a data field for a `TransactionManager`.

8. Uncomment the lines of code in the `@BeforeEach` method that create Mockito mocks for:
   a. `TransactionManager`
   b. `EmployeeManagementServiceProxy`

9. Write a test to verify the `startTransaction`, `commitTransaction` methods are called on the `TransactionManager` when `insertNewManager` is called on the proxy.
   a. Use Mockito to do the verification.
   b. Use Mockito to verify the `insertNewManager` method is called on the `EmployeeManagementServiceImpl` when `insertNewManager` is called on the proxy.


«interface»
EmployeeManagementService
+ insertNewManager(Employee, List<Employee>): Employee
EmployeeManagementServiceProxy
- service: EmployeeManagementService
- transactionManager: TransactionManager
+ insertNewManager(Employee, List<Employee>): Employee
EmployeeManagementServiceImpl
+ insertNewManager(Employee, List<Employee>): Employee
TransactionManager
- dataSource: DataSource
+ startTransaction(): Connection
+ commitTransaction(): boolean
+ rollbackTransaction(): boolean


:EmployeeManagementServiceProxy
:TransactionManager
:EmployeeManagementServiceImpl
:EmployeeDao
insertNewManager(Employee, List<Employee>): Employee
startTransaction(): Connection
insertNewManager(Employee, List<Employee>): Employee
insertEmployee(Employee): Employee
:manager
loop
[for each employee]
updateEmployee(Employee): Employee
:manager
commitTransaction(): boolean
:manager

Congratulations! You have completed this exercise.

Optional Exercise 10.4: Writing an Integration Test for the Service and DAO

Since the `EmployeeManagementServiceImpl` class does not start a transaction directly, it is possible to test the insert, update, and delete operations and then roll the changes back in the `@AfterEach` method as you have done before.

Be sure to start a transaction in the `@BeforeEach` method and to roll back the transaction in the `@AfterEach` method.

You can start and roll back the transaction as was done in the DML tests.

1. Open the `ServiceDaoIntegrationTest.java` file.
   a. Declare a `TransactionManager` field.

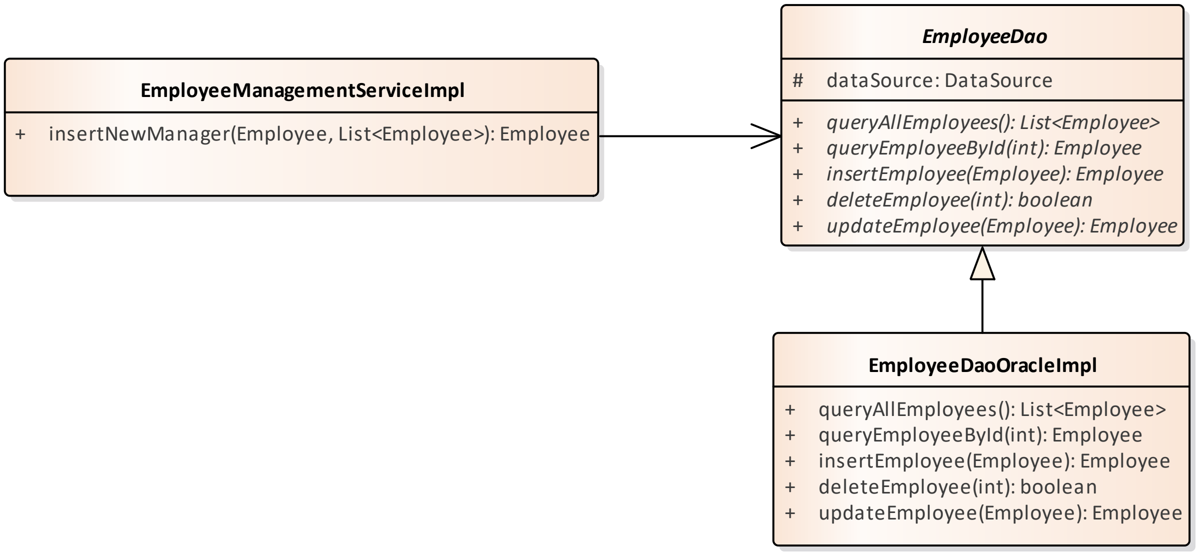
2. Do the following in the `@BeforeEach` method:
   a. Create an `EmployeeDaoOracleImpl` and assign to the `dao` field.
   b. Create an `EmployeeManagementService` and assign to the `service` field.
   c. Add a `TransactionManager` field to the test class, then create a `TransactionManager` and assign to the `TransactionManager` field.
   d. Use the `TransactionManager` to start a transaction.

3. Do the following in the `@AfterEach` method:
   a. Use the `TransactionManager` to roll back the transaction.

4. Use TDD to implement the `insertNewManager()` method. First, write an integration test case.
   f. `insertNewManager()` takes two arguments: the manager (an `Employee` instance) and a list of `Employees` who report to the manager.
   g. The test case should verify the database contents before and after the call to `insertNewManager()`.

5. `EmployeeManagementService insertNewManager()` should call methods of the `EmployeeDao` to perform the necessary database operations.
   a. Insert the new manager.
   b. Insert all the manager's employees.
   c. If the DAO throws an exception during any database operation, `insertNewManager()` should throw a `ServiceException`.

6. Now run the integration test and verify that you get a green bar.


EmployeeManagementServiceImpl
+ insertNewManager(Employee, List<Employee>): Employee
EmployeeDao
# dataSource: DataSource
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
+ insertEmployee(Employee): Employee
+ deleteEmployee(int): boolean
+ updateEmployee(Employee): Employee
EmployeeDaoOracleImpl
+ queryAllEmployees(): List<Employee>
+ queryEmployeeById(int): Employee
+ insertEmployee(Employee): Employee
+ deleteEmployee(int): boolean
+ updateEmployee(Employee): Employee

Congratulations! You have completed this exercise.

This page was intentionally left blank.

Chapter 11: Aggregating Information

Exercise 11.1: Using the Aggregate Functions

Connect to the `SCOTT` account.

1. Write a query displaying how many rows there are in the `emp` table.

```
     Count
----------
        14
```

2. Write a query displaying the empno, name, salary, and commission for all rows in the `emp` table, sequencing the list in salary (ascending) order.

```
     EMPNO ENAME             SAL       COMM
---------- ---------- ---------- ----------
      7369 SMITH             800
      7900 JAMES             950
      7876 ADAMS            1100
      7521 WARD             1250        500
      7654 MARTIN           1250       1400
      7934 MILLER           1300
      7844 TURNER           1500          0
      7499 ALLEN            1600        300
      7782 CLARK            2450
      7698 BLAKE            2850
      7566 JONES            2975
      7788 SCOTT            3000
      7902 FORD             3000
      7839 KING             5000
```

3. Write a query displaying how many non-null salary values exist and how many distinct non-null salary values exist in the `emp` table.

```
     Count          CDistinct
---------- ------------------
        14                 12
```

4. Write a query displaying how many non-null commission values exists, the sum of the non-null commission values, the average of the non-null commission values for all rows in the `emp` table.

```
      Count        Sum    Average
----------- ---------- ----------
          4       2200        550
```

5. Modify the above query by adding the average of commission values treating unknown values as zero. Round this value to three decimal places.

```
     Count        Sum    Average Average of all Records
---------- ---------- ---------- ----------------------
         4       2200        550                157.143
```

6. Write a query displaying the largest and smallest salaries in the `emp` table.

```
Maximum Salary Minimum Salary
-------------- --------------
          5000            800
```

7. Write a query displaying the latest and the earliest hire dates in the `emp` table.

```
Maximum Hire Date Minimum Hire Date
----------------- -----------------
12-JAN-1983       17-DEC-1980
```

Congratulations! You have completed this exercise.

Exercise 11.2: GROUP BY and HAVING

Connect to the `HR` account.

1. For each department in the `employees` table, show the total count of employees, the highest salary, the smallest salary, the sum of the salaries, and the average of salaries (round to the nearest whole currency unit).

```
DEPARTMENT_ID   COUNT(*) MIN(SALARY) MAX(SALARY) Total Salary Avg Salary
------------- ---------- ----------- ----------- ------------ ----------
           10          1        4400        4400         4400       4400
           20          2        6000       13000        19000       9500
           30          6        2500       11000        24900       4150
           40          1        6500        6500         6500       6500
           50         45        2100        8200       156400       3476
           60          5        4200        9000        28800       5760
           70          1       10000       10000        10000      10000
           80         34        6100       14000       304500       8956
           90          3       17000       24000        58000      19333
          100          6        6900       12008        51608       8601
          110          2        8300       12008        20308      10154
                       1        7000        7000         7000       7000
```

Note: Your output may not be in the same sequence.

2. Modify the presentation sequence of the above query: the departments should be in ascending average salary order.

```
DEPARTMENT_ID   COUNT(*) MIN(SALARY) MAX(SALARY) Total Salary Avg Salary
------------- ---------- ----------- ----------- ------------ ----------
           50         45        2100        8200       156400       3476
           30          6        2500       11000        24900       4150
           10          1        4400        4400         4400       4400
           60          5        4200        9000        28800       5760
           40          1        6500        6500         6500       6500
                       1        7000        7000         7000       7000
          100          6        6900       12008        51608       8601
           80         34        6100       14000       304500       8956
           20          2        6000       13000        19000       9500
           70          1       10000       10000        10000      10000
          110          2        8300       12008        20308      10154
           90          3       17000       24000        58000      19333
```

3. Modify the previous query by adding a new column: calculate how much each department's smallest salary is below the average salary. Sequence the list by this expression.

```
DEPARTMENT_ID   COUNT(*) MIN(SALARY) MAX(SALARY) Total Salary Avg Salary  Below Avg
------------- ---------- ----------- ----------- ------------ ---------- ----------
           20          2        6000       13000        19000       9500       3500
           80         34        6100       14000       304500       8956       2856
           90          3       17000       24000        58000      19333       2333
          110          2        8300       12008        20308      10154       1854
          100          6        6900       12008        51608       8601       1701
           30          6        2500       11000        24900       4150       1650
           60          5        4200        9000        28800       5760       1560
           50         45        2100        8200       156400       3476       1376
           40          1        6500        6500         6500       6500          0
           70          1       10000       10000        10000      10000          0
           10          1        4400        4400         4400       4400          0
                       1        7000        7000         7000       7000          0
```

4. Modify the above query by changing the analysis: we now want to know all the above information by the manager each employee works for.

```
MANAGER_ID   COUNT(*) MIN(SALARY) MAX(SALARY) Total Salary Avg Salary  Below Avg
---------- ---------- ----------- ----------- ------------ ---------- ----------
       100         14        5800       17000       155400      11100       5300
       101          5        4400       12008        44916       8983       4583
       148          6        6100       11500        51900       8650       2550
       149          6        6200       11000        50000       8333       2133
       147          6        6200       10500        46600       7767       1567
       146          6        7000       10000        51000       8500       1500
       145          6        7000       10000        51000       8500       1500
       121          8        2100        4200        25400       3175       1075
       108          5        6900        9000        39600       7920       1020
       103          4        4200        6000        19800       4950        750
       122          8        2200        3800        23600       2950        750
       123          8        2500        4000        25900       3238        738
       120          8        2200        3200        22100       2763        563
       124          8        2500        3500        23000       2875        375
       114          5        2500        3100        13900       2780        280
       102          1        9000        9000         9000       9000          0
                    1       24000       24000        24000      24000          0
       201          1        6000        6000         6000       6000          0
       205          1        8300        8300         8300       8300          0
```

5. Another analysis request has been made: modify the previous query to “rate” managers within each department by how far their lowest employee salary is below average.

```
DEPTID  MGRID  COUNT(*) MIN(SALARY) MAX(SALARY) Total Salary Avg Salary  Below Avg
------ ------ --------- ----------- ----------- ------------ ---------- ----------
    80    148         6        6100       11500        51900       8650       2550
    80    149         5        6200       11000        43000       8600       2400
    80    100         5       10500       14000        61000      12200       1700
    80    147         6        6200       10500        46600       7767       1567
    80    146         6        7000       10000        51000       8500       1500
    80    145         6        7000       10000        51000       8500       1500
    50    100         5        5800        8200        36400       7280       1480
    50    121         8        2100        4200        25400       3175       1075
   100    108         5        6900        9000        39600       7920       1020
    60    103         4        4200        6000        19800       4950        750
    50    122         8        2200        3800        23600       2950        750
    50    123         8        2500        4000        25900       3238        738
    50    120         8        2200        3200        22100       2763        563
    50    124         8        2500        3500        23000       2875        375
    30    114         5        2500        3100        13900       2780        280
    10    101         1        4400        4400         4400       4400          0
    90                1       24000       24000        24000      24000          0
    30    100         1       11000       11000        11000      11000          0
   110    205         1        8300        8300         8300       8300          0
    60    102         1        9000        9000         9000       9000          0
   100    101         1       12008       12008        12008      12008          0
    90    100         2       17000       17000        34000      17000          0
    20    100         1       13000       13000        13000      13000          0
    20    201         1        6000        6000         6000       6000          0
   110    101         1       12008       12008        12008      12008          0
    70    101         1       10000       10000        10000      10000          0
    40    101         1        6500        6500         6500       6500          0
          149         1        7000        7000         7000       7000          0
```

6. Modify the above query to show only those managers within a department that have more than 5 employees reporting to them.

```
DEPTID  MGRID  COUNT(*) MIN(SALARY) MAX(SALARY) Total Salary Avg Salary  Below Avg
------ ------ --------- ----------- ----------- ------------ ---------- ----------
    80    148         6        6100       11500        51900       8650       2550
    80    147         6        6200       10500        46600       7767       1567
    80    146         6        7000       10000        51000       8500       1500
    80    145         6        7000       10000        51000       8500       1500
    50    121         8        2100        4200        25400       3175       1075
    50    122         8        2200        3800        23600       2950        750
    50    123         8        2500        4000        25900       3238        738
    50    120         8        2200        3200        22100       2763        563
    50    124         8        2500        3500        23000       2875        375
```

Bonus Exercise (if time permits)

7. Display the sum of salary, the average of salary, and the number of employees in departments, consolidating departments 0-99 together, 100-199 together, etc.

```
Depts by 100s SUM(SALARY) AVG(SALARY)   COUNT(*)
------------- ----------- ----------- ----------
            0      612500        6250         98
          100       71916      8989.5          8
                     7000        7000          1
```

8. Display the average of all departments’ average salaries. Round the result to whole currency units.

```
Avg of Dept Avgs
----------------
            8153
```

9. Compare the result from the step above to the average of employee salaries. Is it the same? Why or why not?

Congratulations! You have completed this exercise.

Exercise 11.3: Using Subqueries

Connect to the `HR` account.

Using subqueries, write queries to display the following information:

1. Display the department id and department name for all departments that have one or more employees. Order the result by `department_id`.

```
DEPARTMENT_ID DEPARTMENT_NAME
------------- ------------------------------
           10 Administration
           20 Marketing
           30 Purchasing
           40 Human Resources
           50 Shipping
           60 IT
           70 Public Relations
           80 Sales
           90 Executive
          100 Finance
          110 Accounting
```

2. Display the employee id, first name, last name, and salary for all employees that have a salary greater than the average salary for all employees. Order the result by salary in descending sequence.

```
EMPLOYEE_ID FIRST_NAME      LAST_NAME            SALARY
----------- --------------- ---------------- ----------
        100 Steven          King                  24000
        101 Neena           Kochhar               17000
        102 Lex             De Haan               17000
        145 John            Russell               14000
        146 Karen           Partners              13500
        201 Michael         Hartstein             13000
        205 Shelley         Higgins               12008
        108 Nancy           Greenberg             12008
        147 Alberto         Errazuriz             12000
        168 Lisa            Ozer                  11500
        148 Gerald          Cambrault             11000
        174 Ellen           Abel                  11000
        114 Den             Raphaely              11000
        162 Clara           Vishney               10500
        149 Eleni           Zlotkey               10500
        150 Peter           Tucker                10000
        156 Janette         King                  10000
        204 Hermann         Baer                  10000
        169 Harrison        Bloom                 10000
        170 Tayler          Fox                    9600
        163 Danielle        Greene                 9500
        157 Patrick         Sully                  9500
        151 David           Bernstein              9500
```

Continued on next page

```
        158 Allan           McEwen                 9000
        109 Daniel          Faviet                 9000
        103 Alexander       Hunold                 9000
        152 Peter           Hall                   9000
        175 Alyssa          Hutton                 8800
        176 Jonathon        Taylor                 8600
        177 Jack            Livingston             8400
        206 William         Gietz                  8300
        110 John            Chen                   8200
        121 Adam            Fripp                  8200
        153 Christopher     Olsen                  8000
        120 Matthew         Weiss                  8000
        159 Lindsey         Smith                  8000
        122 Payam           Kaufling               7900
        112 Jose Manuel     Urman                  7800
        111 Ismael          Sciarra                7700
        154 Nanette         Cambrault              7500
        160 Louise          Doran                  7500
        171 William         Smith                  7400
        172 Elizabeth       Bates                  7300
        164 Mattea          Marvins                7200
        161 Sarath          Sewall                 7000
        155 Oliver          Tuvault                7000
        178 Kimberely       Grant                  7000
        113 Luis            Popp                   6900
        165 David           Lee                    6800
        203 Susan           Mavris                 6500
        123 Shanta          Vollman                6500

51 rows selected.
```

3. Display the employee id, first name, last name, and salary for the employee that has the highest salary.

```
EMPLOYEE_ID FIRST_NAME      LAST_NAME            SALARY
----------- --------------- ---------------- ----------
        100 Steven          King                  24000
```

4. Display the employee id, first name, last name, salary, and commission_pct for all employees that have a salary greater than the average salary for all employees and a commission_pct greater than the average commission_pct for all employees. Order the result by last_name.

```
EMPLOYEE_ID FIRST_NAME      LAST_NAME         SALARY COMMISSION_PCT
----------- --------------- --------------- -------- --------------
        174 Ellen           Abel               11000             .3
        151 David           Bernstein           9500            .25
        148 Gerald          Cambrault          11000             .3
        160 Louise          Doran               7500             .3
        147 Alberto         Errazuriz          12000             .3
        152 Peter           Hall                9000            .25
        175 Alyssa          Hutton              8800            .25
        156 Janette         King               10000            .35
        158 Allan           McEwen              9000            .35
        168 Lisa            Ozer               11500            .25
        146 Karen           Partners           13500             .3
        145 John            Russell            14000             .4
        161 Sarath          Sewall              7000            .25
        159 Lindsey         Smith               8000             .3
        157 Patrick         Sully               9500            .35
        150 Peter           Tucker             10000             .3
        162 Clara           Vishney            10500            .25

17 rows selected.
```

Bonus Exercise (if time permits)

5. Display the employee id, first name, and last name for the employee(s) that work in London. You will need to use two levels of subquery.

```
EMPLOYEE_ID FIRST_NAME           LAST_NAME
----------- -------------------- -------------------------
        203 Susan                Mavris
```

Congratulations! You have completed this exercise.

Chapter 12: Set Operators

Exercise 12.1: Set Operators

Connect to the `HR` account.

1. Produce a short report showing the number of employees who earn commission and the number who do not.

   a. Your report should look like this:

```
Type                                      Count
------------------------------------ ----------
Employees who earn commission                35
Employees who do not earn commission         72
```

   b. Use a set operator to create this report.

Congratulations! You have completed this exercise.

This page intentionally left blank.

Chapter 13: Programming with PL/SQL

Exercise 13.1: Building Anonymous Blocks

Connect to the `HR` account.

1. In the declaration section, declare a record `emp_rec` that uses the `employees` table as a basis.

   Use `%ROWTYPE`.

2. In the executable section, retrieve information for employee with a last name of `'Austin'` into `emp_rec`.

3. In the executable section, write a conditional structure that sets the new salary based on the employee’s commission.

```
if commission_pct is undefined or zero then
  increase salary by a flat $500
if commission_pct is less than .2 then
  increase salary by $300
otherwise
  increase salary by $100
```

4. Write an `UPDATE` statement that sets the employee’s salary to the value derived in the `IF` statement.

5. Select from the `employees` table for employee Austin to view the salary prior to running the block.

6. Execute the block. Again, query the `employees` table to check your results.

```
LAST_NAME       SALARY
--------------- ----------
Austin          5300
```

7. Test your code with other employees. Use Lee and then King. Validate your results by viewing the before and after values.

8. Roll back your changes.

9. Enhance your PL/SQL block by using a `CASE` statement. Execute the block and check your results. Test with other employees as in Step 6.

10. Roll back your changes.

Bonus Exercise (if time permits)

11. Enhance the block by adding exception handling. Use the block which uses the `CASE` statement. The select statement may return no rows, one row, or many rows. Add the `EXCEPTION` section and add a handler for each of the possible errors.

12. Good practice recommends that you also add a handler for any other error that may occur. If unexpected conditions occur, raise an error, display “Contact support” and append the Oracle error message.

13. Test the program by running it for different last names. Use Austin, Smith, and Howard.

14. Roll back your changes.

Congratulations! You have completed this exercise.

Exercise 13.2: Using Cursors

Connect to the `HR` account.

In this exercise, you will declare and use a cursor to process the records in the `employees` table. Increase employee salary by $5,000 for employees who were hired before Jan. 1, 2003. The cursor will accept one parameter, which limits the query to return only the required employees.

1. In the declaration section, declare a cursor that selects an employee record from the `employees` table. The cursor should accept one parameter called `in_date_hired`. Compare `in_date_hired` with the `date_hired` column in the `WHERE` clause of the `SELECT` statement. The cursor should include a `FOR UPDATE` clause to lock the selected rows.

2. Declare a record to hold the cursor results. Also, declare a numeric variable, `raise`, and initialize it to 5000. Finally, declare a date variable, `v_date`, and initialize it to Jan. 1, 2003.

3. In the executable section, open the cursor and pass `v_date` as a parameter.

4. Use a simple `LOOP…END LOOP` construct that loops through each record that the cursor returns.

5. Within the `LOOP`, use a `FETCH` statement to retrieve the row into the cursor record.

6. Add an `EXIT WHEN` statement to exit the loop when no more rows are returned by the cursor.

   Warning! An exit statement must be included to avoid an endless loop.

7. Add an `UPDATE` statement, which uses the current cursor row.

8. Execute the block and check your results. Make sure to check records that should not have been updated as well!

```
EMPLOYEE_ID FIRST_NAME      LAST_NAME        HIRE_DATE     SALARY
----------- --------------- ---------------- ----------- --------
        203 Susan           Mavris           07-JUN-2002    11500
        204 Hermann         Baer             07-JUN-2002    15000
        205 Shelley         Higgins          07-JUN-2002    17008
        206 William         Gietz            07-JUN-2002    13300
        102 Lex             De Haan          13-JAN-2001    22000
        108 Nancy           Greenberg        17-AUG-2002    17008
        109 Daniel          Faviet           16-AUG-2002    14000
        114 Den             Raphaely         07-DEC-2002    16000

8 rows selected.
```

9. Roll back your changes.

Bonus Exercise (if time permits)

10. Modify the previous example to use a `FOR-LOOP` cursor instead of a regular cursor with a `LOOP`. ***Set*** the salary to $11000; i.e., not a raise.

```
EMPLOYEE_ID FIRST_NAME           LAST_NAME                 HIRE_DATE       SALARY
----------- -------------------- ------------------------- ----------- ----------
        203 Susan                Mavris                    07-JUN-2002      11000
        204 Hermann              Baer                      07-JUN-2002      11000
        205 Shelley              Higgins                   07-JUN-2002      11000
        206 William              Gietz                     07-JUN-2002      11000
        102 Lex                  De Haan                   13-JAN-2001      11000
        108 Nancy                Greenberg                 17-AUG-2002      11000
        109 Daniel               Faviet                    16-AUG-2002      11000
        114 Den                  Raphaely                  07-DEC-2002      11000

8 rows selected.
```

11. Roll back your changes.

Congratulations! You have completed this exercise.

Chapter 14: Creating Stored Procedures, Functions, and Packages

Exercise 14.1: Stored Procedures, Functions, and Packages

Connect to the `HR` account.

In this exercise, you will create and execute a procedure that updates an employee if one exists. Otherwise, assume this is a new employee and insert a new employee into the table.

1. Use the `CREATE PROCEDURE` statement to define a procedure called `update_emp`. It requires six input parameters: `parm_employee_id`, `parm_last_name`, `parm_email, parm_hire_date, parm_job_id` and `parm_salary`.

2. In the executable section, update the salary in the `employees` table for the specified employee_id. Verify that the last name, hire date, and job_id match the input parameters before making the change.

3. If no rows were updated because the input employee id does not exist in the `employees` table, then insert the input data into the table as a new row.

4. Complete the procedure with an END statement.

5. Store the procedure in the database by executing the CREATE PROCEDURE statement.

   If you get a warning that the procedure was created with compilation errors, use the `SHOW ERRORS` command.

6. Test the procedure by creating a simple PL/SQL block that calls the procedure and passes parameters to it. Use employees Chen and Johnston for your tests. Check your results.

7. Roll back your changes.

Bonus Exercise (if time permits)

Change the procedure from the previous exercise to perform an `INSERT` only if the department the employee is assigned to presently has a manager.

We will use a separate function to perform the test. The function is used only by the procedure. Therefore, it can be hidden by defining it as a private function in a package.

8. Start by dropping the independent procedure update_emp since we now want to include it into a package.

9. Write a package specification, pack_employee that contains a declaration of the update_emp procedure. Use the CREATE PACKAGE statement. You will need to add one more parameter to pass in a department id.

10. Use the procedure from previous section as the basis for this procedure specification. Remember to add the additional parameter for department id. Remove the procedure body, which starts with the keyword `IS` and finishes with an `END` statement.

11. Submit the package specification to the database for compilation and storage.

12. Write the package body for pack_employee using the `CREATE PACKAGE BODY` statement.

13. Write a function definition in the package body. The function should return the manager id for the assigned department or a null value if a manager is not assigned. The function will require a single parameter for the `department_id`.

14. The function should be defined before the procedure.

15. Use a `FUNCTION` definition statement. The `FUNCTION` consists of a `SELECT` statement and a `RETURN` statement.

16. Copy the procedure from the previous section and make the following changes:
    a. Add an additional parameter to pass in the `department_id`.
    b. Add an `IF–THEN` construct around the `INSERT` statement so that it inserts only if the department being assigned to has a manager assigned.
    c. Use a `PROCEDURE` definition statement instead of the `CREATE PROCEDURE` statement within the package body.

17. The package body must be completed with an `END` statement.

18. Submit the package body to the database for compilation and storage.

19. Test the package by executing the packaged procedure from an anonymous PL/SQL block. Check your results.

Congratulations! You have completed this exercise.

Chapter 15: Testing PL/SQL

Exercise 15.1: Writing PL/SQL Tests with utPLSQL

Connect to the `HR` account.

In this exercise, you will create a function that checks whether a salary is appropriate for a given job. You will work TDD, writing tests using utPLSQL.

1. Start by creating a test package for your function.
   a. Name it appropriately.
   b. Annotate it with the `--%suite` annotation.

2. In the package specification, declare your first test specification.
   a. Your test specification is a procedure annotated with `--%test`.
   b. This will be a test for the normal behavior of the function.

3. Now create the package body for your test package.
   a. Create the definition of the test specification.
   b. It will pass in a known `job_id` and a `salary` in the right range, expecting boolean `TRUE`.
   c. The body will not compile because the function under test does not exist.

4. Write the function.
   a. It should accept two parameters, the `job_id`, and `salary`.
   b. It should return `BOOLEAN`.
   c. For now, it should just return `TRUE`.

5. Execute all tests in the current schema.
   a. Turn on server output (`SET SERVEROUTPUT ON`).
   b. Write an anonymous block that executes `ut.run()`. You can optionally pass in your test package name to ensure only that package runs.
   c. One test will run, and it should pass.

6. Now write a second test in the test package.
   a. It should be a test for a known `job_id` and a `salary` below the `min_salary`. It should expect `FALSE`.
   b. Remember to add it to the package specification with the correct annotation.

7. Run the tests again. The new test should fail.

8. Now implement the appropriate functionality to return `FALSE` when the `salary` is below `min_salary` for the given `job_id` and `TRUE` otherwise.
   a. For now, ignore exceptional situations, such as the `job_id` not existing, just focus on making the tests pass.

9. Run the tests and see that they pass.

10. Complete the normal behavior by testing for `FALSE` when the salary is above the `max_salary` for the given `job_id`.
    a. Run the test, it should fail.
    b. Write the functionality.
    c. Re-run the test and repeat the cycle until it passes.
    d. Refactor if necessary.

11. Now start to add some negative tests and implement the functionality. Write a single new test each time and then write the functionality to make it pass. Your function should behave like this:
    a. If `job_id` or `salary` are `NULL`, it should raise an exception.
    b. If `job_id` does not exist, it should return `FALSE`. Depending on how you wrote the function, you may not need to add any code to make this happen but add a test for it anyway.
    c. If you think of any other exceptional situations, determine appropriate behavior and write a test.

12. Consider adding some additional boundary condition tests.

Congratulations! You have completed this exercise.

Exercise 15.2: Testing Updates With utPLSQL

Connect to the `HR` account.

In this exercise, you will create a procedure that updates a salary for an employee if the salary is appropriate for their job.

1. Start by creating a test package for your function.
   a. Name it appropriately.
   b. Annotate it with the `--%suite` annotation.

2. In the package specification, declare your first test specification.
   a. Your test specification is a procedure annotated with `--%test`.
   b. This will be a test for the normal behavior of the procedure.

3. Now create the package body for your test package.
   a. Create the definition of the test specification.
   b. It will pass in a known `employee_id` and a `salary` in the right range for their `job_id`, expecting a single row to be updated.
   c. The body will not compile because the procedure under test does not exist.

4. Write the procedure.
   a. It should accept two parameters, `employee_id` and `salary`.
   b. For now, it should just update the row anyway.

5. Execute your new test package.
   a. Turn on server output (`SET SERVEROUTPUT ON`).
   b. Write an anonymous block that executes `ut.run()`. Pass in your test package name to ensure only that package runs.
   c. One test will run, and it should pass.

6. Now write a second test in the test package.
   a. It should be a test where the `salary` is below the `min_salary`. It should expect no rows to be updated.
   b. Remember to add it to the package specification with the correct annotation.

7. Run the tests again. The new test should fail.

8. Now implement the appropriate functionality to ensure the `salary` is only updated when it is above the `min_salary` for the `job_id` of the employee.
   a. If you wish, you can use the function you created in the previous exercise. Writing it as a single `UPDATE` is possible but a little harder.
   b. For now, ignore exceptional situations, such as the `employee_id` not existing, just focus on making the tests pass.

9. Run the tests and see that they pass.

10. Complete the normal behavior by testing for no rows being updated when the salary is above the `max_salary` for the given `job_id` of the employee.
    a. Run the test, it should fail. If you used the function in the previous steps, it may already pass. This is not a problem: the test is still valuable to protect against regression errors.
    b. Write the functionality, if necessary.
    c. Re-run the test and repeat the cycle until it passes.
    d. Refactor if necessary.
    e. Consider refactoring the tests to reduce the amount of repeated code.

11. Now start to add some negative tests and implement the functionality. Write a single new test each time, and then write the functionality to make it pass. Your procedure should behave like this:
    a. If `employee_id` or `salary` is `NULL`, it should raise an exception.
    b. If `employee_id` does not exist, it should simply not update any data, but it should not throw an unexpected exception (e.g., `NO_DATA_FOUND`).
    c. If you think of any other exceptional situations, determine appropriate behavior and write a test.

Congratulations! You have completed this exercise.

Chapter 16: Creating Triggers

Exercise 16.1: Working with Triggers

Connect to the `HR` account.

In this exercise, you will write a trigger that ensures new employees are only inserted if their salary is in the right range for their job. You will use the function you created in Exercise 15.1: if you are concerned about your solution to that exercise, take the function from the solution file.

1. Check that your function has been created and that all the tests pass.

2. Create a test for inserting a new employee, expecting failure.
   a. Note that in this case, since we are testing a trigger, the `INSERT` will occur in the test code.
   b. Use a known unused value for the `employee_id` and `PU_CLERK` for the `job_id`. Choose a `salary` outside the right range. Set the other mandatory columns to reasonable values.
   c. The `INSERT` should fail with an exception.

3. Run your test.
   a. It should fail since the `INSERT` will succeed.

4. Now create an insert trigger for the `employees` table.
   a. Use the `CREATE TRIGGER` statement.
   b. This should be a `AFTER` trigger.
   c. This trigger should be fired each time a row is inserted.
   d. For now, it should just throw the exception your test is expecting.

5. Your test should now pass, but clearly, no data can be inserted into `employees`!

6. Now create a second test—this time inserting the same employee with a salary in the right range.
   a. The test should expect the insert to succeed.

7. Run your tests. The new test should fail since the `INSERT` fails.

8. Now write the correct body of the trigger.
   a. It should use the function to decide whether to allow the `INSERT`.
   b. Use the `:NEW` pseudo-record to get the appropriate data values.

9. Run the tests again until they succeed.

10. When you are done, drop your trigger.

Bonus Exercise (if time permits)

11. Modify your trigger, so it also works for an `UPDATE` of `job_id` or `salary`.
    a. Have it throw a different exception when performing an `UPDATE`.
    b. Work TDD.

12. Drop your trigger.

Congratulations! You have completed this exercise.

Chapter 17: Data Definition Language

Exercise 17.1: Table Management

Connect to the HR account.

In this exercise, you will create a new table, a sequence, and a view. Using these, you will explore various DDL commands and their effects.

1. Create a table called benefits.
   a. Use the following columns definitions:

```
benefit_id             NUMBER(3)    NOT NULL
benefit_name           VARCHAR2(25)
benefit_type           VARCHAR2(20) DEFAULT 'HEALTH CARE'
benefit_effective_date DATE
benefit_max_allowance  NUMBER(8,2)
```

   b. Make benefit_id the primary key.

2. Describe the benefits table to verify the definition.

3. Create a sequence called seq_benefits. Make its starting and incremental values 1.

4. Insert a row into the benefits table *without* a column list.
   a. Use the sequence for the benefit_id.
   b. Make the name "401k", the type "Retirement", set the effective date to Jan. 1, 2010, and the max allowance to 250,000.

5. Insert another row into the benefits table *with* a column list, specifying *all* columns.
   a. Use the sequence for the benefit_id.
   b. Make the name "Medical PPO", the type "Health", set the effective date to Jan. 1, 2011, and the max allowance to 100,000.

6. Insert another row into the benefits table *with* a column list, specifying *all* columns.
   a. Use the sequence for the benefit_id.
   b. Set the type to the reserved word DEFAULT.
   c. Make the name "Medical Ins", set the effective date to Jan. 1, 2012, and the max allowance to 125,000.

7. Display all the rows in the benefits table. What is the value of type for the 3rd row?

8. Insert another row into the benefits table with a column list. Specify all column names except for benefit_type.
    a. Use the sequence for the benefit_id.
    b. Make the name "No default name provided", set the effective date to Jan. 1, 2013, and the max allowance to 150,000.

9. Display all the rows in the benefits table. What is the value of type for the 4th row?

10. Update all benefits rows whose type value begins with "H" to the table DEFAULT.

11. Display all the rows in the benefits table. What is the value of the type columns?

12. COMMIT the changes.

13. Create a view called "vw_h_b" that contains the benefit ID, name, type, and max allowance from the benefits table. Only allow the rows whose value for type begins with "HEALTH".

14. Describe this view.

15. Display all the rows through the view.

16. Try to add a new, numeric, mandatory column to the benefits table: max_dependents. Why did the attempt fail?

17. Try to add the column again, this time specifying a DEFAULT value of 0.

18. Display the benefits table: what value is in the max_dependents column?

19. Re-run the select through the view. Does it include the new column?

Bonus Exercise (if time permits)

20. Modify the maximum size of the benefit_name column to be 50. Does this succeed?
    a. Describe the benefits table to see the impact of the command.

21. Try to modify the maximum size of the benefit_name column to be 20. Why does this fail?

22. Insert into the benefits table by selecting all the rows from the benefits table.
    a. Use the row values for all columns except for the benefit_id: use the sequence number for this value.

23. Display all the rows in the benefits table. How many are there now?

24. Issue a `ROLLBACK`.

25. Re-run the previous set insert.
    a. Insert into the `benefits` table by selecting all the rows from the `benefits` table.
    b. Use the row values for all columns except for the `benefit_id`: use the sequence number for this value.

26. Display all the rows in the `benefits` table. How many are there now? What are the benefit IDs? Can you explain their values?

Congratulations! You have completed this exercise.

This page intentionally left blank.

Chapter 20: Amazon DynamoDB

Exercise 20.1: Access DynamoDB Using AWS CLI

In this exercise, you will use a local version of DynamoDB and the AWS Command Line Interface to explore basic DynamoDB actions. You will first create a simple table and then add some data to it, before using various actions to query the data.

1. Use Visual Studio Code for this exercise.
   a. Unzip the file `dynamodb.zip` into a folder on your desktop.
      i. *Note:* running exercise from any other than the `C:\` drive will NOT work.
   b. Open that folder in VS Code.

2. Open a terminal window and examine `start.bat`.
   a. This file runs the local version of DynamoDB.
   b. Run the file by entering: `.\start.bat` (*Note:* do not forget "`.\`".)
   c. Leave that terminal window open.

3. Open a new terminal window to use for the rest of the exercise.

4. Run the command: `aws configure`.
   a. Do not change the Access Key or Secret Access Key, instead just press `Return`.
   b. Enter a suitable region, choose `us-east-1`, `eu-west-1` or `ap-south-1`, depending on your location.
   c. You can also press return and leave the output format unset.

5. Run the command:

```
aws dynamodb create-table --generate-cli-skeleton input
```

6. Compare the output from that command to the contents of the file `CreateTable.json`.
   a. The output of the previous command is a pro forma for the `CreateTable` action.
   b. The file `CreateTable.json` creates a table called `Music` with a two-part key (`artist` and `songTitle`).

7. Run the following command and check the output:

```
aws dynamodb create-table \
    --endpoint-url http://localhost:8000 \
    --cli-input-json file://CreateTable.json
```

   a. Replace the backslash (`\`) with the appropriate line continuation character for your terminal or type the command on one line.
   b. Make sure not to forget the `--endpoint-url` parameter or the `file://` protocol on the `--cli-input-json` parameter.
   c. Normally, you would need to wait for the table to become active, but since we are using the local version of DynamoDB, it becomes active immediately.

8. Run these commands to see your new table:

```
aws dynamodb list-tables \
    --endpoint-url http://localhost:8000
```

```
aws dynamodb describe-table \
    --endpoint-url http://localhost:8000 \
    --table-name Music
```

9. Run the command:

```
aws dynamodb put-item --generate-cli-skeleton input
```

   a. Again, the output is a pro forma for the `PutItem` action.
   b. Note that this pro forma contains a number of legacy items. Consult the documentation to see which items those are: https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_PutItem.html

10. Add some data to your table:

```
aws dynamodb put-item \
    --endpoint-url http://localhost:8000 \
    --cli-input-json file://PutItem1.json
```

```
aws dynamodb put-item \
    --endpoint-url http://localhost:8000 \
    --cli-input-json file://PutItem2.json
```

   a. Note that the two files do not set the same attributes on both items.

11. Try different ways of retrieving data (review each file before you run it):

```
aws dynamodb get-item \
    --endpoint-url http://localhost:8000 \
    --cli-input-json file://GetItem.json
```

```
aws dynamodb query \
    --endpoint-url http://localhost:8000 \
    --cli-input-json file://Query.json
```

```
aws dynamodb scan \
    --endpoint-url http://localhost:8000 \
    --cli-input-json file://Scan.json
```

Bonus Exercise (if time permits)

12. Try other variations of these commands; for example, you can also insert an item using this version of the `PutItem` action:

```
aws dynamodb put-item \
    --endpoint-url http://localhost:8000
    --table-name Music \
    --item file://item.json \
    --return-values ALL_OLD
```

a. It won’t return any data unless you run it twice because the contents of `item.json` are a new item.

b. You could also specify the item at the command line in either JSON or shorthand format.

13. Using your own data, add new items to the `Music` table and write `GetItem`, `Query`, and `Scan` commands to retrieve them.

Congratulations! You have completed this exercise.

Exercise 20.2: Java Document API

In this exercise, you will access DynamoDB from Java code and investigate how to achieve the same basic tasks using the Java Document API.

1. Make sure the local version of DynamoDB is running, as described at the start of the previous exercise.

2. Use Eclipse for the rest of the exercise.

3. Open the `DynamoDB` project in the Relational Databases workspace.

4. Open `DynamoDbDocumentDao` and `DynamoDbDocumentDaoTest`.
   a. Review the tests and corresponding code.
   b. Compare them to the files used for the previous exercise.

5. Run the tests. They should all pass.

6. Take a copy of `music.json`.
   a. Do not edit the existing file since it is used by a number of test classes.
   b. Change the contents of your new JSON file to reflect some of your own musical choices.
   c. Edit `DynamoDbDocumentDaoTest` to refer to the new file.
   d. Make the tests pass with your new data.

Bonus Exercise (if time permits)

7. Open `DynamoDbLowLevelDao` and `DynamoDbDaoLowLevelTest`. These use the low-level API.
   a. Review the code and compare them to the Document versions.
   b. Especially look at features that you know to be different, such as waiting for asynchronous operations, converting to and from Java classes, and iterating over windowed collections.

Congratulations! You have completed this exercise.

Exercise 20.3: Java Object Mapper

In this exercise, you will continue accessing DynamoDB using Java, but this time you will access data using the Java Object Mapper. You will see how to achieve tasks such as inserting and querying data using this higher-level interface.

1. Make sure the local version of DynamoDB is running, as described at the start of the previous exercise.
2. In the `DynamoDB` project, open `DynamoDbMapperDao` and `DynamoDbMapperDaoTest`.
   a. Review the tests and corresponding code.
   b. Compare them to the files used for the previous exercise.
3. Run the tests. They should all pass.
4. Take a copy of `music.json`.
   a. Do not edit the existing file since it is used by a number of test classes.
   b. Change the contents of your new JSON file to reflect some of your own musical choices.
   c. Edit `DynamoDbMapperDaoTest` to refer to the new file.
   d. Make the tests pass with your new data.

Congratulations! You have completed this exercise.