Fidelity INVESTMENTS

WORKING WITH RELATIONAL DATABASES

EXERCISE MANUAL

ROITRAINING
MAXIMIZE YOUR TRAINING INVESTMENT

INNOVATION STARTS HERE. Leap

This page 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 properly.
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.

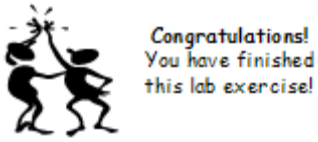
14. Now, execute both statements together as a script (<F5>). Note that the results of both statements are displayed in the Script Output.

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.



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 finished
this lab 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.

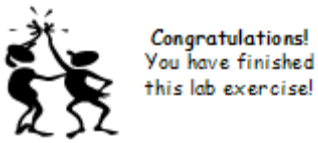
```
STREET_ADDRESS                         Street Name
-------------------------------------- --------------------------------------
8204 Arthur St                         Arthur St
6092 Boxwood St                        Boxwood St
93091 Calle della Testa                Calle della Testa
2004 Charade 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
```



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.

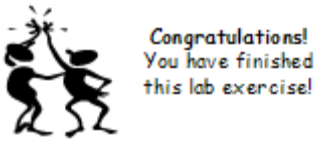
```
EMPLOYEE_ID FIRST_NAME           LAST_NAME            JOB_ID
----------- -------------------- -------------------- ----------
        101 Neena                Kochhar              AC_MGR
        101 Neena                Kochhar              AC_ACCOUNT
        102 Lex                  De Haan              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
```



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.

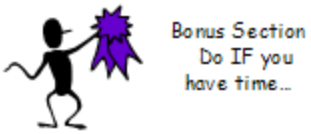
	JOB_TITLE	EMPLOYEE_ID
1	President	(null)
2	Administration Vice President	(null)
3	Sales Manager	176

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)



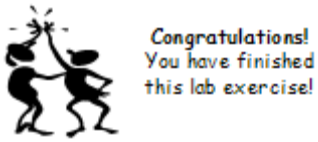
6. Joining the `employees`, `job_history`, and `jobs` tables, display the job title, employee ID, start date, and employee first and last names for ALL employees, whether or not they have any job history.

   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.



Chapter 5: Additional SQL Functions

Exercise 5.1: Additional SQL Functions

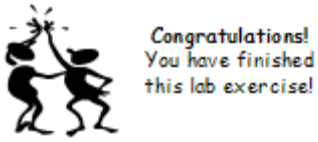
Connect to the `HR` account.

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
```



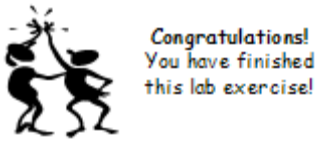
This page intentionally left blank.

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.



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 create a simple Java class to connect to the database.

Over the course of the next two days, you will enhance this class to be a Data Access Object (DAO). We will cover the specifics of DAOs a little later, but for now it is enough to know that it is a class that centralizes database access.

1. Open the `ScottEmp` project in Eclipse and review the `Employee` class. This is the Java representation of the `scott.emp` table.

2. You will need the Oracle database driver jar in your classpath. Add the appropriate dependency to your `pom.xml`.

```
<dependency>

<groupId>com.oracle.database.jdbc</groupId>

<artifactId>ojdbc8</artifactId>

<version>18.3.0.0</version>

</dependency>
```

3. Now create a new Java class to hold the database access code.
   a. Name it EmployeeDao.
   b. Put it in an appropriate package (in all the solution projects, we will use `com.fidelity.integration` to show that it is our integration layer).

4. Now create a test class for the `EmployeeDao`.
   a. Right-click `EmployeeDao` 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 `EmployeeDaoTest` and it is in the same package as `EmployeeDao`.
   d. Change the source folder to the test source folder.
   e. Add any jar files that may be suggested.

5. Create a test for the connection.
   a. You will need an instance of the `EmployeeDao` class. You may create this in the test method, or in an `@BeforeEach` method.
   b. Call `employeeDao.getConnection()`, which should return a `Connection` object.
   c. Test that the `Connection` is not `null`.
   d. Because `getConnection()` doesn't exist yet, you will get an error and the method name will be underlined in red.
   e. Hover over the method name and take the option that offers to create the method.
   f. Save the empty method and run your test, which should fail because the `Connection` is null.

6. Write the `getConnection()` method.
   a. Have it make a connection to the database and return the connection object. *Note:* View the `db.properties` file in the resources folder to see the correct properties to use for connecting to the Oracle database.
   b. Ideally, it would check whether you are already connected and only connect if necessary.
   c. The method should be public for now.
   d. You may hard-code the connection parameters.
   e. For now, it is acceptable for your method to throw any checked exceptions rather than catching them.
   f. Re-run your test and see that it works.

Optional steps, if you have time:

In a normal Java application, we would not hard-code the database connection parameters. We would read them from a properties file.

7. A suitable file already exists in the project resources folder. It is named `db.properties`.
   a. Examine the parameters and see that they correspond to what you have already been using. (Note that there are no spaces on the lines: spaces can lead to errors when reading the properties.)
   b. This file is on the classpath: you can use it without saying how to find it.

8. Modify your `getConnection()` to read from the properties file:
   a. Add the following code:

```
Properties properties = new Properties();
properties.load(this.getClass().getClassLoader()
    .getResourceAsStream("db.properties"));
String dbUrl = properties.getProperty("db.url");
String user = properties.getProperty("db.username");
String password = properties.getProperty("db.password");
```

b. If you prefer, you can include the `db.driver`.

c. This code may throw `IOException`, you need to handle that somehow.

Exercise 7.2: Creating Objects from Database Query

In this exercise, you will create methods in your DAO class to access employees from the `scott.emp` table.

1. Create a method to retrieve all records from the `scott.emp` table and return a list of `Employee` objects.
   a. Work in TDD fashion. Start by identifying the tests you need. How will you tell whether the method returned the right set of employees?
   b. Once you have written the test, let Eclipse create the method for you.
   c. Connect wherever you like. You may connect in each test, or in each call to the new method. You should position `close`() statements appropriately according to where you have chosen to connect.
   d. Note that `Employee.hiredate` is a String. You should treat it accordingly.
   e. 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.

2. 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. Again, work TDD fashion. What tests do you need? Write the tests and then let Eclipse create the method for you.
   b. 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.

Optional steps, if you have time:

3. Create a method to retrieve a single `scott.emp` record by `empno` and return a single `Employee` object.

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

Exercise 7.3: Creating Secure Database Queries

In this exercise, you will use PreparedStatement to handle queries with parameters.

1. Re-factor your method from Step 2 of the previous exercise to use PreparedStatement instead of Statement.
   a. Again, work in TDD fashion. You should have all the tests you need. You should be able to re-factor your method and use the tests to check that you have done it correctly.
   b. If you did not manage to do Step 2, start from scratch and work in a TDD fashion.
   c. Add a test that proves that injection is not possible. Consider the string `"SMITH' OR '1' = '1"`.

Optional steps, if you have time:

2. Review all the code you have written so far and look for re-factoring opportunities.
3. Create a method to retrieve a single `scott.emp` record by `empno` and return a single `Employee` object.
   a. If you did the optional step in the previous exercise, this is a re-factoring exercise.
4. Create a method to retrieve all the `scott.emp` records in a particular department and return a list of `Employee` objects.
   a. If you did the optional step in the previous exercise, this is a re-factoring exercise.

Exercise 7.4: Creating a DAO

In this exercise, you will re-factor the code you have written so far in this chapter and work towards the DAO pattern.

1. It helps to have the DAO pattern in mind when re-factoring.
   a. We have already created the DAO class as a place to encapsulate all the database communication.
   b. Clients interacting with the DAO should not be aware of the database structure, or even that communication is with a database.
   c. Methods should accept Java objects and primitives as parameters and return types. `ResultSet` should never be seen outside the DAO.
   d. The DAO should take care of connecting to the database.
   e. It is acceptable for the DAO to have an explicit `close()` method that must be called when we are finished.

2. If you haven't already, ensure that the `Connection` object is a field of the DAO and is set by the `getConnection()` method.

3. Re-factor the DAO class to create an explicit `close()` method:
   a. The method should attempt to close the `Connection` if it is not `null` (in other words if the `getConnection()` method has been called).
   b. It should set the field to `null` to indicate that it no longer contains a valid connection.
   c. Call `close()` from an `@AfterEach` method in your test class.

4. Re-factor the DAO class so that each query method (each method that uses some sort of SQL statement) calls the `getConnection()` method before getting the statement.
   a. In this way, there will always be a valid connection before it is needed.
   b. Change the connection method to private and remove any direct tests.
   c. Your individual methods should not close the connection, that should only occur in the DAO's close method. But they should close the `Statement` or `PreparedStatement` that they use: re-factor the code to use try-with-resources blocks.

5. Ensure each query method accepts and returns only business objects (preferred), or primitives.

6. Ensure that you are handling exceptions properly by catching them and throwing a sub-class of RuntimeException.

Chapter 8: Testing with Databases

Exercise 8.1: Writing JDBC Unit Tests

Review all your JDBC unit tests so far to ensure that they are correctly testing for the right outcomes.

Treat the `scott` schema as a set of known data (in other words, assume you can rely on the data to be the same for each test).

Make changes to the data and confirm that your tests fail as expected. Remember that if you are making changes through SQL Developer, you will need to commit those changes before they can be seen by your Java code.

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

Exercise 8.2: Creating a Mock DAO

In this exercise, you will create a mock `EmployeeDao`.

1. Use the **Refactor** menu to **Rename…** `EmployeeDao` to `EmployeeDaoOracleImpl` or some similar name.
   a. Also rename the `EmployeeDaoTest` class to be in line with the new name of the class under test.
2. Use the **Refactor** menu to **Extract Interface…**
   a. The interface should contain all the public methods of the DAO.
   b. Name it `EmployeeDao`.
3. Create a new class that implements the `EmployeeDao` interface.
   a. Name it `EmployeeDaoMockImpl`, or similar.
   b. Create a copy of the test class that applies the same tests to the mock implementation. Name it appropriately.
   c. Run the tests and confirm that they all fail.
4. Implement the methods of the mock DAO such that the tests pass.
   a. The methods should not communicate with a database. They should create and return data as needed.
   b. You do not need to create a complete simulation of the DAO. It is sufficient for the tests to pass.
   c. If your tests are checking the number of records returned, you may change the counts in the tests to a representative number (e.g., 2 – 4) that covers the other test cases, rather than creating a mock that returns 14 records. However, you should work TDD (change the test first).

Chapter 9: Updating Databases

Exercise 9.1: Writing to a Database

In this exercise, you will create methods to insert and update data through the DAO.

1. Working in a TDD fashion, create a method in the real DAO implementation to update employee data.
   a. Update all non-key fields to new values.
   b. Your tests should be non-destructive (each test should reset the data after checking the results).
   c. Your method should work with an `Employee` object rather than primitives.
2. Again, working in a TDD fashion, create a method to insert a new employee.
   a. How will you test this non-destructively?

Optional steps, if you have time:

3. Create a new method to perform a selective bulk update. For example, give all employees a raise if they started before a certain date.
4. Create methods that use batches to make multiple updates and inserts from lists of `Employee` objects.

Exercise 9.2: Using Transactions

In this exercise, you will practice using manual transaction control.

1. Create a method that will insert a collection of employees in a transaction.
   a. What might cause different parts of this process to fail?
   b. What if a new employee already exists?
   c. Will you validate, or will you attempt to detect and roll back?
   d. Can you simulate other database problems? If so, how?

Note: you could use a batch to do this, but for this exercise, use a manual transaction instead.

Chapter 10: Advanced JDBC

Exercise 10.1: Using `BigDecimal` in JDBC

1. Re-factor the `Employee` class and the DAO so that salary and commission are `BigDecimal` rather than `double`.
   a. Handle `NULL` correctly, at least for commission.

Optional steps, if you have time:

2. Extend null handling to the manager field.

Exercise 10.2: Using `LocalDate` in JDBC

1. Re-factor the `Employee` class and the DAO so that hiredate is a `LocalDate` rather than `String`.
   a. Handle `NULL` correctly.

Exercise 10.3: Test Using Transactions

1. Re-factor the DAO and DAO tests so that, as far as possible, all tests are run in a transaction and rolled back at the end.
   a. Are there any tests or methods that will give you trouble? How will you test these?

This page 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 finished
this lab 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

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?

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.
```

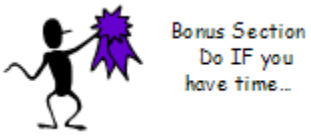
3. Display the employee id, first name, last name, and salary for the employee that has the highest salary.

```
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.
```



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
```

Chapter 12: Set Operators

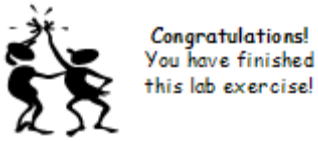
Exercise 12.1: Set Operators

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.



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.

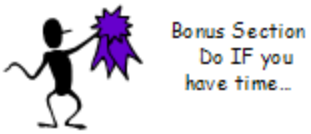
```
LAST_NAME       SALARY
--------------- ----------
Austin          5300
```

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

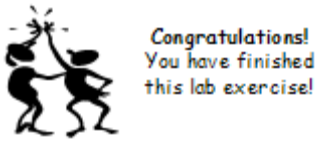
8. Roll back your changes.

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.



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.



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.

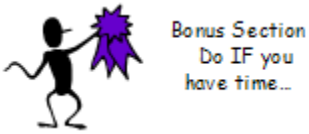
7. Add an `UPDATE` statement, which uses the current cursor row.

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.

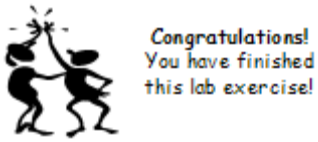


10. Modify the previous example to use a `FOR-LOOP` cursor instead of a regular cursor with a `LOOP`. ***Set*** the salary to $11000; i.e., not a raise.

```
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.



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 prior to 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.

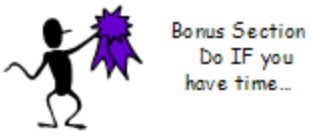
4. Complete the procedure with an END statement.

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.



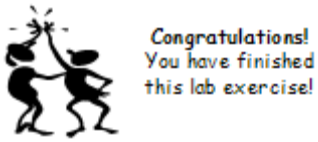
Change the procedure from the previous exercise to perform an `INSERT` only if the department the employee is assigned to presently has a manager.

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.



This page intentionally left blank.

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.

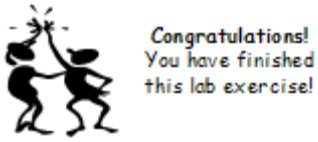
8. Now implement the appropriate functionality to return `FALSE` when the `salary` is below `min_salary` for the given `job_id` and `TRUE` otherwise.
   a. For now, ignore exceptional situations, such as the `job_id` not existing, just focus on making the tests pass.

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.



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.

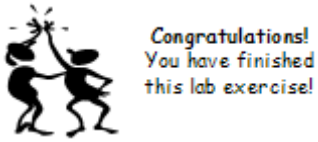
7. Run the tests again. The new test should fail.

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` are `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.



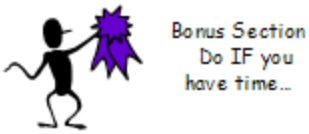
Chapter 16: Creating Triggers

Exercise 16.1: Working with Triggers

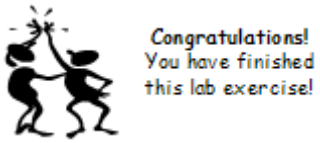
Connect to the `HR` account.

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.



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.



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.

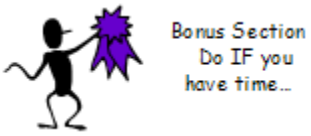
4. Insert a row into the `benefits` table *without* a column list.
   a. Use the sequence for the `benefit_id`.
   b. Make the name "401k", the type "Retirement", set the effective date to Jan. 1, 2010, and the max allowance to 250,000.

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.

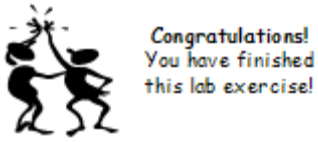
7. Display all the rows in the `benefits` table. What is the value of type for the 3rd row?

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?



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. Rerun 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?



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
```

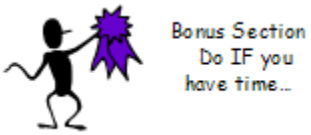
   a. Note that the two files do not set the same attributes on both items.

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
```

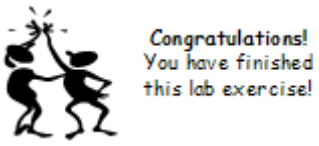


12. Try other variations of these commands; for example, you can also insert an item using this version of the `PutItem` action:

```
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.

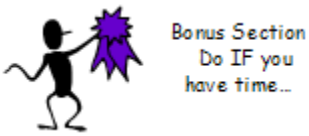
13. Using your own data, add new items to the `Music` table and write `GetItem`, `Query`, and `Scan` commands to retrieve them.



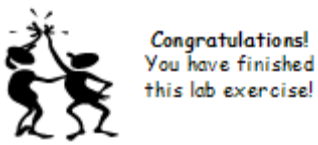
Exercise 20.2: Java Document API

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.



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.



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.