Database Processing

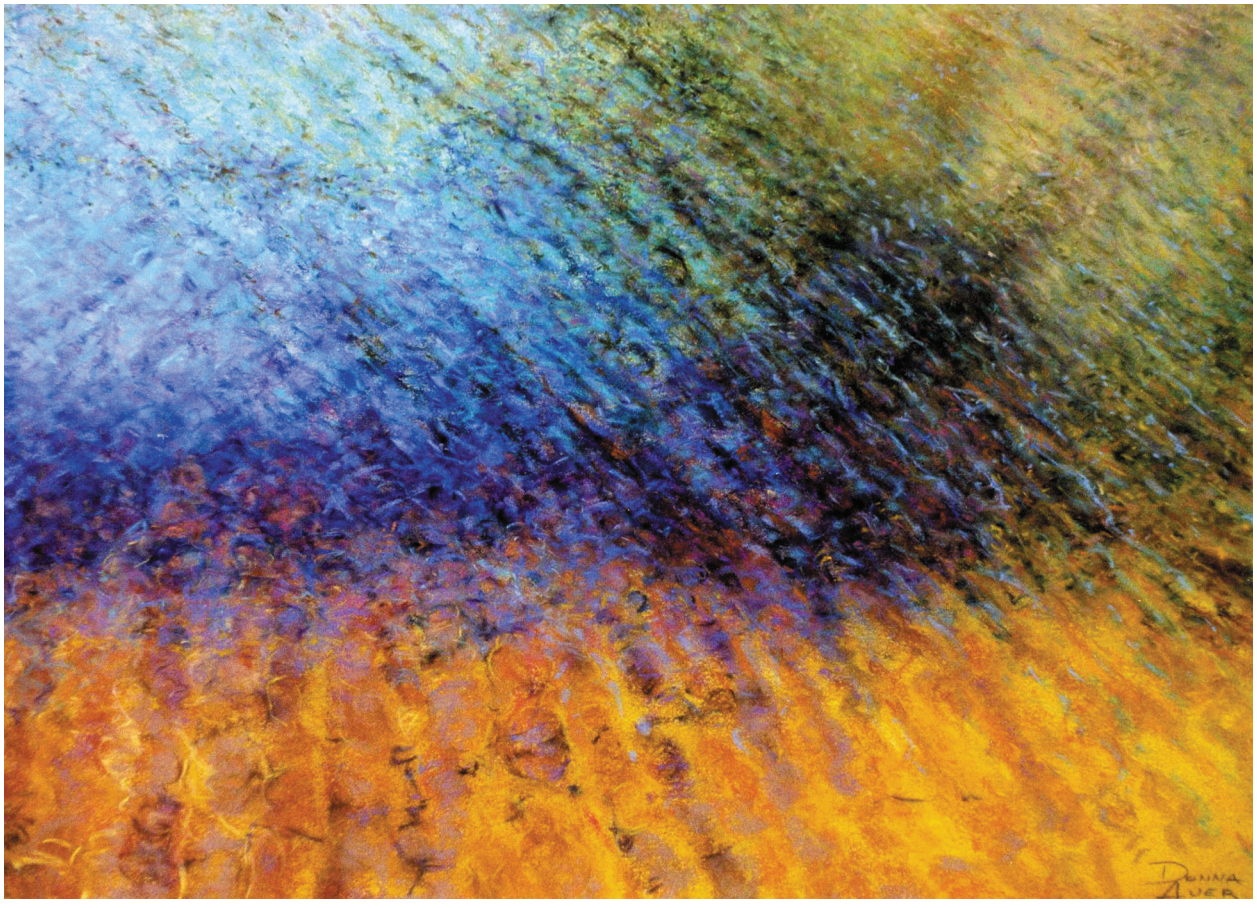
Fundamentals, Design, and Implementation

14th Edition

David M. Kroenke • David J. Auer

Online Chapter 10B

Managing Databases with Oracle Database



Vice President, Business Publishing: Donna Battista
Editor in Chief: Stephanie Wall
Acquisitions Editor: Nicole Sam
Program Manager Team Lead: Ashley Santora
Program Manager: Denise Weiss
Editorial Assistant: Olivia Vignone
Vice President, Product Marketing: Maggie Moylan
Director of Marketing, Digital Services and Products: Jeanette Koskinas
Executive Product Marketing Manager: Anne Fahlgren
Field Marketing Manager: Lenny Ann Raper
Senior Strategic Marketing Manager: Erin Gardner
Product Marketing Assistant: Jessica Quazza
Project Manager Team Lead: Jeff Holcomb
Project Manager: Ilene Kahn
Operations Specialist: Diane Peirano
Senior Art Director: Janet Slowik
Text Designer: Integra Software Services Pvt. Ltd.
Cover Designer: Integra Software Services Pvt. Ltd.
Cover Art: Donna Auer
Vice President, Director of Digital Strategy & Assessment: Paul Gentile
Manager of Learning Applications: Paul Deluca
Digital Editor: Brian Surette
Digital Studio Manager: Diane Lombardo
Digital Studio Project Manager: Robin Lazrus
Digital Studio Project Manager: Alana Coles
Digital Studio Project Manager: Monique Lawrence
Digital Studio Project Manager: Regina DaSilva
Full-Service Project Management and Composition: Integra Software Services Pvt. Ltd.
Printer/Binder: RRD Willard
Cover Printer: Phoenix Color/Hagerstown
Text Font: 10/12 Mentor Std Light

Library of Congress Cataloging-in-Publication Data

Kroenke, David M.
Database processing: fundamentals, design, and implementation/David M. Kroenke, David J. Auer.–Fourteenth edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-387670-3 (student edition)–ISBN 978-0-13-387676-5
(instructor's review copy)
1. Database management. I. Auer, David J. II. Title.
QA76.9.D3K76 2016
005.74–dc23

2015005632

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 10: 0-13-387670-5
ISBN 13: 978-0-13-387670-3

10B

Managing Databases with Oracle Database

Chapter Objectives

- To install Oracle Database and create a database
- To use Oracle Database 12*c*'s Web-based Database Control Enterprise Manager utility
- To use Oracle Database's graphical utilities
- To create and use Oracle Database tablespaces
- To understand how Oracle Database implements server and database security
- To be able to submit both SQL DDL and DML via the SQL Developer utility
- To import Microsoft Excel data into a database table
- To understand the use of SQL/Persistent Stored Modules (SQL/PSM) in Oracle Database PL/SQL
- To understand the purpose and role of user-defined functions and learn how to create simple user-defined functions
- To understand the purpose and role of stored procedures and learn how to create simple stored procedures
- To understand the purpose and role of triggers and learn how to create simple triggers
- To understand how Oracle Database implements indexes, concurrency control, and cursors
- To understand the fundamental features of Oracle Database backup and recovery facilities

This chapter describes the basic features and functions of Oracle Database.[1] The discussion uses the View Ridge Gallery database from Chapter 7, and it parallels the discussion of the database administration tasks in Chapter 9. The presentation is similar in scope and orientation to those for SQL Server 2014 in Chapter 10A and for MySQL 5.6 in Chapter 10C. However, among the DBMS products we are discussing, Oracle Oracle Database is the proverbial *horse of a different color*; thus, we will have to rearrange our discussion of the topics we cover. Specifically, we will have to discuss the creation of user accounts early in the chapter and show how users are linked to Oracle Database *tablespaces*.

Oracle Database is a very popular DBMS, and it has a long history of development and use. Oracle Database exposes much of its technology to the developer; consequently, it can be tuned and tailored in many ways. All of this means, however, that Oracle Database can be difficult to install and daunting to learn. A gauge of Oracle Database's breadth is that one of the most popular references, *Oracle*

[1] Scott Vandenberg of Siena College contributed material to this chapter.

Database 12c: The Complete Reference by Kevin Loney,[2] is more than 890 pages long, but it does not contain everything about Oracle Database 12*c*. Moreover, techniques that work with a version of Oracle Database on one operating system may need to be altered when working with a version on a different operating system. You will need to be patient with Oracle Database and with yourself and not expect to master this subject overnight.

The Oracle Database 12*c* program suite has many configurations. To start, there are different versions of Oracle Database 12*c*. In addition, there are Forms and Reports, Oracle Designer, and a host of tools for publishing Oracle databases on the Web. For an overview of these products, go to the Oracle Web site Products and Services page at *www.oracle.com/products/index.html*. Add to this the need for Oracle's products to operate on many different operating systems and over networks using several different communication protocols, and you can see why they are so difficult to learn.

Installing Oracle Database

Oracle Corporation's **Oracle Database 12*c*** is an enterprise-class DBMS that has been around for many years. It is important to distinguish between the name of the company, *Oracle* (or more completely *Oracle Corporation*), and the name of the product, *Oracle Database*. In earlier years, the Oracle Database DBMS was the main product produced by Oracle, and the word *Oracle* referred to both the company and the DBMS. Thus, in 2001, Oracle 9*i* was released, followed by Oracle 10*g* (with *g* for *grid*, a reference to grid computing) and Oracle Database 11*g*. As this book goes to print, the current version of Oracle Database is Oracle Database 12*c* (with *c* for *cloud computing*).[3]

As Oracle Corporation acquired more product lines, however, there was a need to specifically refer to the DBMS in the product name, and the DBMS was renamed *Oracle Database*. Oracle Database 11*g* became available in 2007 as Oracle Database 11*g* and later as Oracle Database 11*g* Release 2. Oracle Database 10*g* (10.2) and 11*g* (11.1) are no longer generally available. Oracle Database 12*c* is available in several versions, which can be reviewed at the Oracle Database 12*c* Web site (*www.oracle.com/us/products/database/enterprise-edition/comparisons/index.html*). For our purposes, there are five editions we need to be aware of:

- **Enterprise Edition.** The most powerful and feature-laden version of Oracle Database 12*c*. This version handles as many CPUs and memory as the computer's operating system will support. It has full Data Warehouse Builder features. In addition, OLAP and data Mining (now called Oracle Advanced Analytics) options are available.
- **Standard Edition.** This is the basic commercial version of Oracle Database 12*c*. It does not have the complete feature set of the Enterprise Edition. It can support up to four CPUs and includes only limited data warehouse capabilities.
- **Standard Edition One.** This is a variant of the basic commercial version. It supports only two CPUs and has the same limited data warehouse capabilities of the Oracle Database 12*c* Standard Edition.
- **Express Edition 11*g* Release 2 (Oracle Database XE).** This is a free downloadable version based on Oracle Database 11*g* Release 2, and Oracle documentation refers to it as Oracle Database XE. It has limited features: It supports only one

[2]Bob Bryla and Kevin Loney, *Oracle Database 12c: The Complete Reference* (Berkley, CA: McGraw-Hill Education [Oracle Press], 2014).

[3]For more information on grid computing, see the Wikipedia article at *http://en.wikipedia.org/wiki/Grid_computing*. For information on Oracle grid computing, see *http://docs.oracle.com/cd/E11882_01/server.112/e40540/cmntopc.htm#CNCPT1958*. For information on Oracle cloud computing, see *https://www.oracle.com/cloud/index.html*. For a discussion on the difference between grid computing and cloud computing, see *http://www.brighthub.com/environment/green-computing/articles/68785.aspx*.

CPU, it has 1 GB of memory, and the maximum database size is only 11 GB. Despite its limitations, it is a great learning tool.

- **Personal Edition.** Although not shown on the Web page comparing the previous editions, it is available as an option for purchase on the Oracle Web site at *https://shop.oracle.com/pls/ostore/f?p=dstore:product:0::NO:RP,6:P6_LPI,P6_PROD_HIER_ID:4508888120961805719862,450995828721805720011r*. The Personal Edition 12*c* is available only for Windows and Linux systems. It is intended for single-user use by a developer. It has nearly all the features of the Enterprise Edition.

The Oracle Database Express Edition was introduced with Oracle Database 10*g*, and the current **Oracle Database Express Edition 11g Release 2**, like the SQL Server Express Editions, seems to be designed to compete with MySQL Community Edition (see Chapter 10C). Although MySQL Community Edition does not have as many features as Oracle Database 12*c* or SQL Server 2014, it is an open source database that has had the advantage of being widely available for download over the Internet. It has become widely used and very popular as a DBMS, supporting Web sites running the Apache Web server. MySQL is now owned by Oracle Corporation.

We will be working with Oracle Database 12*c* and Oracle Database Express Edition 11*g* Release 2 in this chapter, but all of our discussion is relevant to the earlier Oracle Database 11*g* Release 2 (11.2). Note, however, that you will still hear the Oracle DBMS product referred to as just *Oracle*, *Oracle 11g Release 2*, or *Oracle 12*c.

Be aware that Oracle Database 12*c* and Oracle Database Express Edition 11*g* Release 2 are enterprise-class DBMS products and, as such, are much more complex than Microsoft Access. Further, the basic DBMS product does not include application development tools, such as form and report generators. Regardless of which version of Oracle Database you are going to use, you should install it now.

BY THE WAY

Choosing which version of Oracle Database to use can be a problem. In this chapter, we describe and discuss Oracle Database 12*c*. However, unless your instructor has already installed this product for your use, you will find it preferable to download and use the **Oracle Database Express Edition 11g Release 2** package (downloadable from *www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html*).

Note that as stated in the Oracle Technical Network (OTN) developer license, there are limits on the hardware characteristics of the computer running the DBMS. Nonetheless, you should be able to do most of the SQL and Web-related database work in this book using Oracle Database Express Edition 11*g* Release 2 and the downloadable Oracle SQL Developer GUI utility (downloadable from *www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html*) discussed later in this chapter [the main exceptions being (1) the Web-based administration of Oracle Database 12*c* using the Oracle Database Control Enterprise Manager utility discussed in this chapter and (2) some of the business intelligence (BI) topics in Appendix J].

Installing a Loopback Adapter

If your computer is set up to get an IP number from a DHCP server (which is typical of high-speed Internet connections), then you must install a *loopback adapter*, which assigns a local and fixed IP address to your computer, before installing Oracle Database 12*c*. (While a loopback adapter is *not* required for Oracle Database Express Edition 11*g* Release 2, it does no harm to have one installed—our installation of the Express Edition included installing a loopback adapter.) It is very important that the loopback adapter be correctly installed *before* the installation of Oracle Database 12*c*, or the Oracle Database installation will have serious problems. See the discussion of Oracle Database 12*c* installation documentation later in this chapter for information on locating the appropriate documentation for your version of Oracle Database 12*c*. If you are using a Windows OS version of Oracle Database 12*c*,

see the Microsoft operating system instructions in Appendix D of the *Oracle Database Installation Guide for Microsoft Windows* at *http://docs.oracle.com/database/121/NTDBI/toc.htm*, section D.4, "Installing a Loopback Adapter."

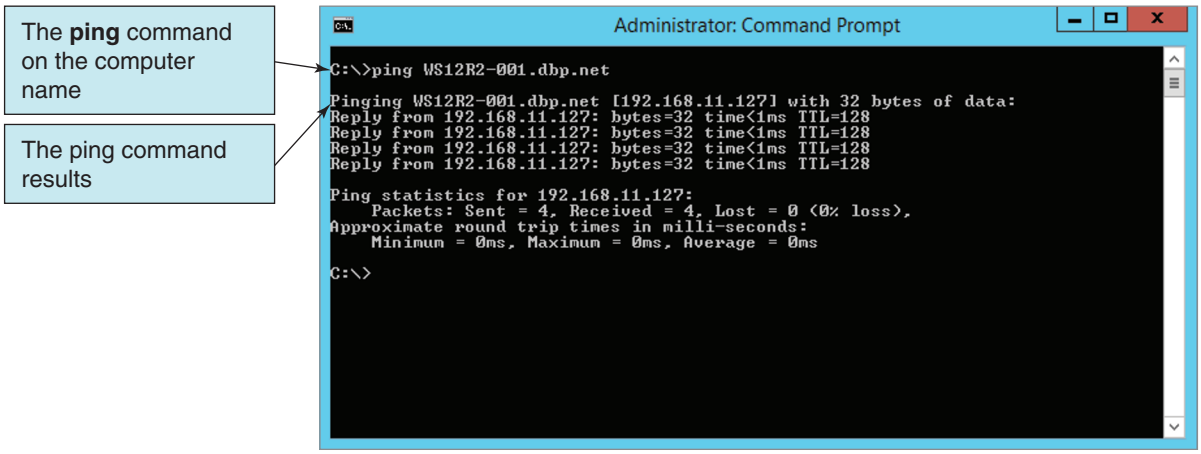
Note that the instructions for installing a loopback adapter include adding a line to your computers HOST file. This line is needed so the Oracle Database 12*c* installation can determine and use the computer name (host name) of your computer during installation. This is an important step, and you need to be sure it is done correctly.

You can test that the loopback adapter, including the line in the HOST file, has been installed correctly by opening a CMD (command line) window (click the **Start** button, type **CMD** in the Search programs and files text box, and press the **Enter** key to start the CMD program). At the command prompt, type ping {your computer name}; for example, we used *ping WS12R2-001*. After the command results are displayed, type ping {your computer name and domain}; for example, we used *ping WS12R2-001.dbp.net*. If both ping commands run correctly, as shown in Figure 10B-1, then the loopback adapter is installed correctly and you are ready to proceed with the installation of Oracle Database 12*c*.

Oracle Database, Java, JavaScript, and the Adobe Flash Player

Oracle Database depends on Java. For example, the Oracle SQL Developer GUI tool (which we will use extensively in this chapter) must be linked to a Java environment before it will run. **Java** is an object-oriented programming language. It was originally developed by Sun Microsystems, but it became an open source product in 2007.[4] Java programs need to be run in the **Java Runtime Environment (JRE)**, which must also be installed on the computer. However, besides the JRE, there is also the **Java Development Kit (JDK)**, which is a **software development kit (SDK)** for Java. The JDK is a more powerful environment than the JRE, and you will need to download and install the JDK for Java Standard Edition (Java SE) in order for SQL Developer to work correctly. You can download the JRE from *www.java.com/en/download/manual.jsp* and the **Java SE Development Kit (JDK)** from *http://java.sun.com/javase/downloads/index.jsp*. If you are or will be working in a Java environment (which is common for Oracle Database users), you may want to download the combination of the JDK and NetBeans. **NetBeans** is an **integrated development environment (IDE)** particularly well suited to Java development. It is also a good IDE for PHP Web page development and is thus an alternative to the Eclipse IDE that we will use in our discussion of PHP Web pages in Chapter 11. Appendix I, "Getting Started with Web Servers, PHP, and the NetBeans IDE," has a discussion of how to install the JRE that you may find useful. At this point, you should download and install the current JRE and JDK (with or without NetBeans) for your computer—we installed the JDK by using the Java SE Development Kit and NetBeans ID Installer as described in Appendix I.

FIGURE 10B-1
Checking the Loopback Adapter Installation



[4]For more information on Java and its history, see the Wikipedia article at *http://en.wikipedia.org/wiki/Java_(programming_language)*.

Oracle Database's Web-based management utilities require that **JavaScript**, a Web page programming language, be enabled in the Web browser.[5] It also requires that the **Adobe Flash Player** be installed.[6] You should set up your Web browser correctly now so there are no problems when you start your work with Oracle Database.

Oracle Database 12*c* Documentation

Oracle provides excellent Oracle Database 12*c* documentation in Web page and PDF format on the Oracle Web site at *http://docs.oracle.com/database/121/nav/portal_11.htm*. Good first references for installing Oracle Database 12*c* are the installation guides (start with the "Quick Installation Guide") for your operating system in the Installing and Upgrading section and the "2-Day DBA" document in the Getting Started section.

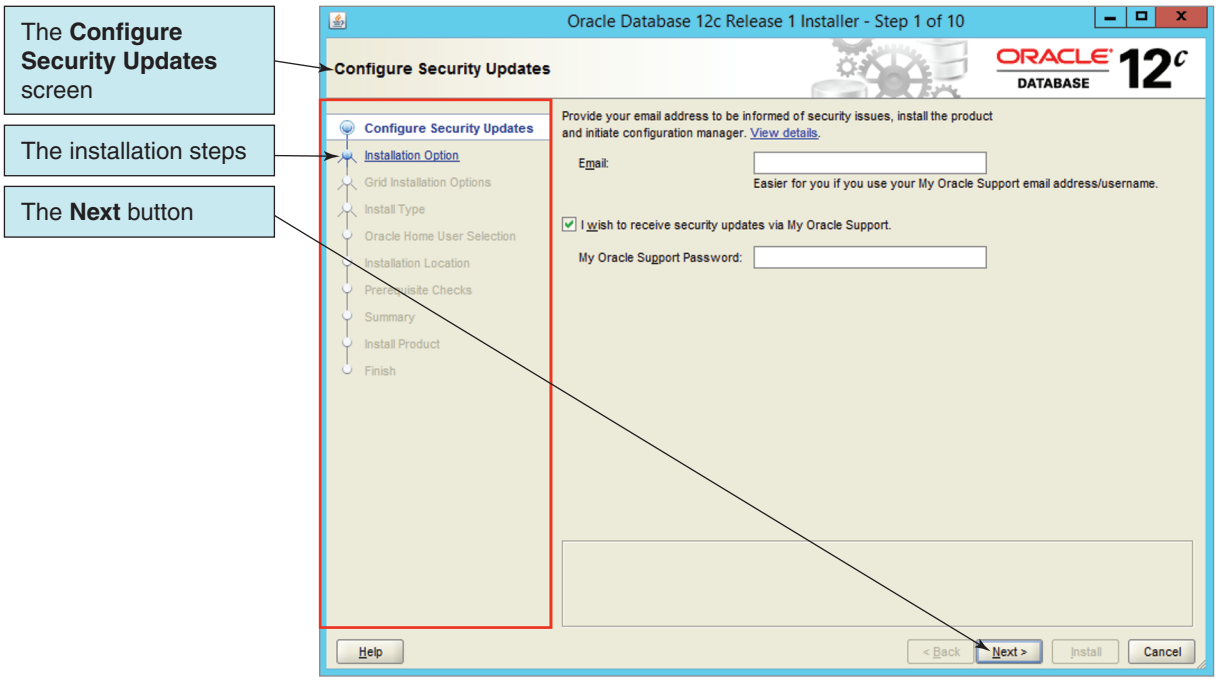
Installing Oracle Database 12c with the Oracle Universal Installer (OUI)

The Oracle documentation refers to using **Oracle Universal Installer (OUI)** to install Oracle Database 12*c*. The OUI is itself installed as part of the installation process—in fact, it is the first thing installed. Figure 10B-2 shows the first OUI screen seen by the user during the installation of Oracle Database 12*c*. After the installation process is complete, the OUI is available by using the Windows Apps icons.[7]

As shown in Figure 10B-2, our installation of the Oracle Database 12*c* Personal Edition starts with a basic user information screen and proceeds through the steps outlined in the left-hand column. An installation of the Personal Edition is identical to an installation of the Enterprise Edition—it is simply intended for the use of one person doing database and application development. For Oracle Database 12*c*, we used the advanced installation method, which is selected on a later screen in the installation process. If you are not using a network installation of Oracle Database 12*c*, a similar installation of the Personal Edition on a workstation will allow you to do everything we discuss in this chapter.

FIGURE 10B-2

The Initial Oracle Universal Installer (OUI) Screen—Oracle Database 12*c*



[5]For instructions on how to enable JavaScript in your Web browser, see the very handy "How to enable JavaScript in your browser" Web page at *www.enable-javascript.com*. For more information about JavaScript, see the Wikipedia article at *http://en.wikipedia.org/wiki/JavaScript*.

[6]To install the Adobe Flash Player, open the Web browser you will be using and go to *http://get.adobe.com/flashplayer*.

[7]The Oracle Database 12*c* DBMS is shown running in Windows Server 2012 R2 in this chapter. All command references are to the Windows Server 2012 R2 commands, and they may vary in other operating systems.

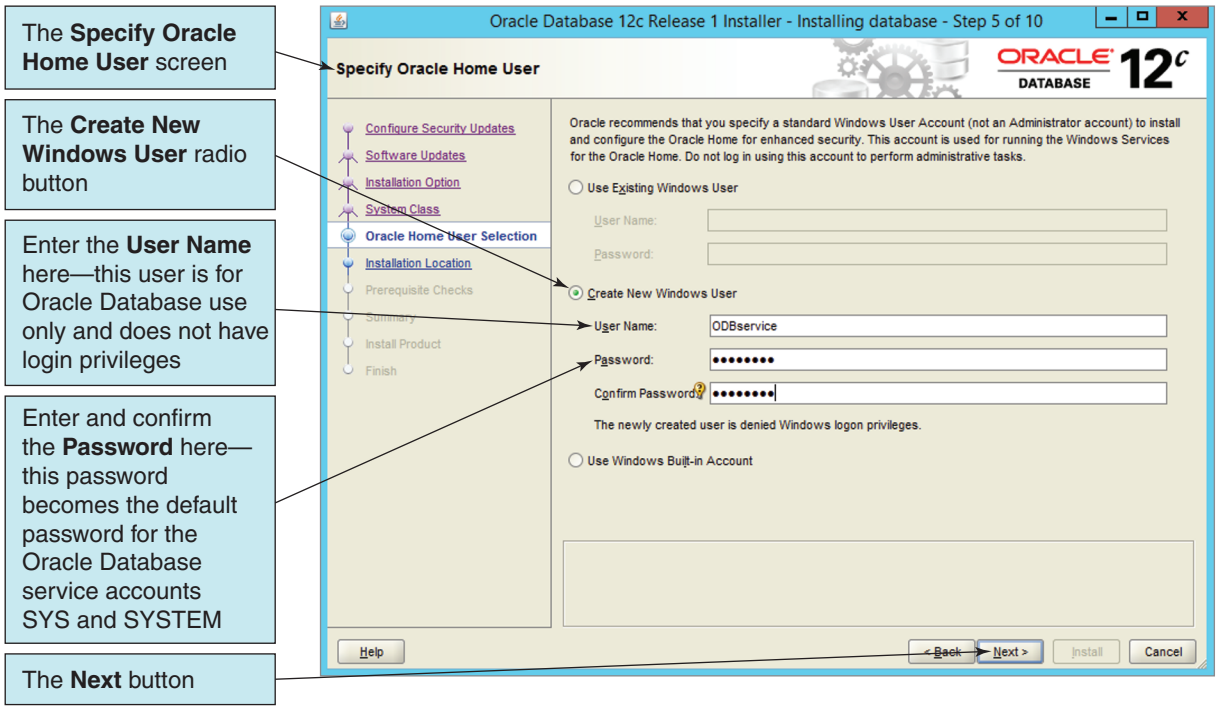


FIGURE 10B-3

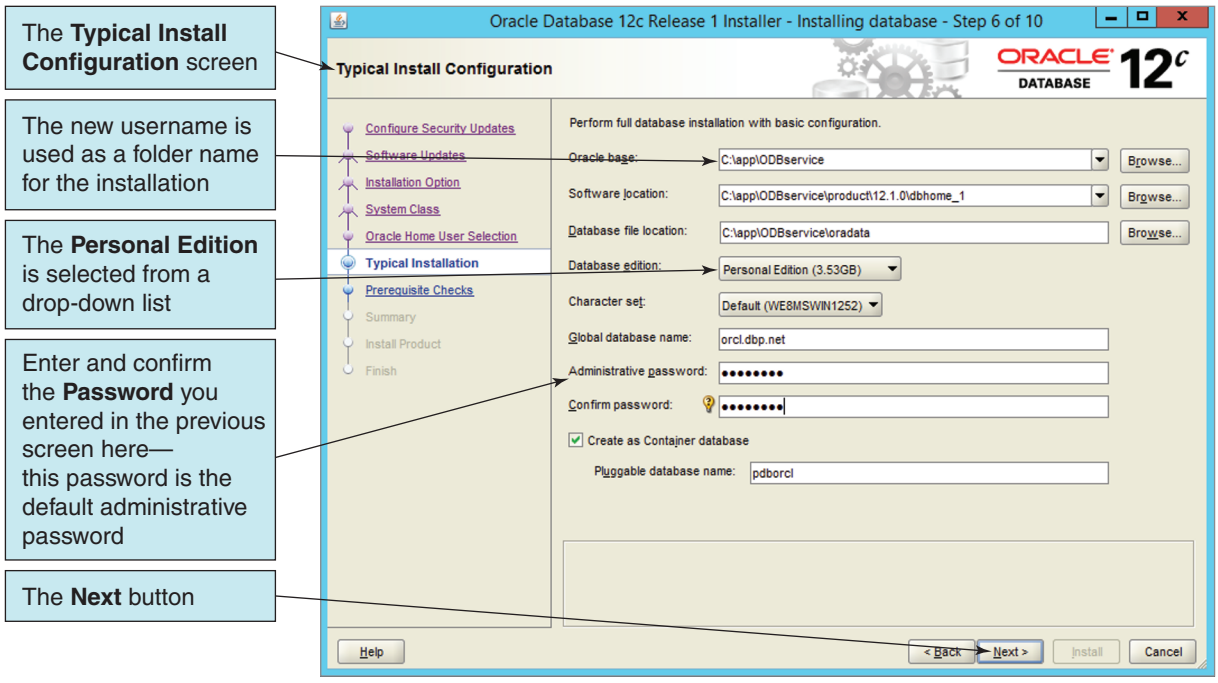
The Specify Oracle Home User Screen

In this setup, the needed Oracle Database SYS and SYSTEM user account passwords are initially set up by creating an Oracle Home User with a password (they can be specifically set later in the process). This is done in the Specify Oracle Home User step of the installation process, as shown in Figure 10B-3. Note that we created a new user for Oracle Database use only (the account has no Windows login privileges), which we have named ODBservice, and provided that user with a password. This is the password that the other Oracle Database system accounts will also use unless changed later in the installation process.

As shown in Figure 10B-4, Oracle Database uses the new username as the *Oracle Base* folder name in the installation process and also requires the use of the new password as the *Administrative Password* for the rest of the installation.

FIGURE 10B-4

The Typical Install Configuration Screen



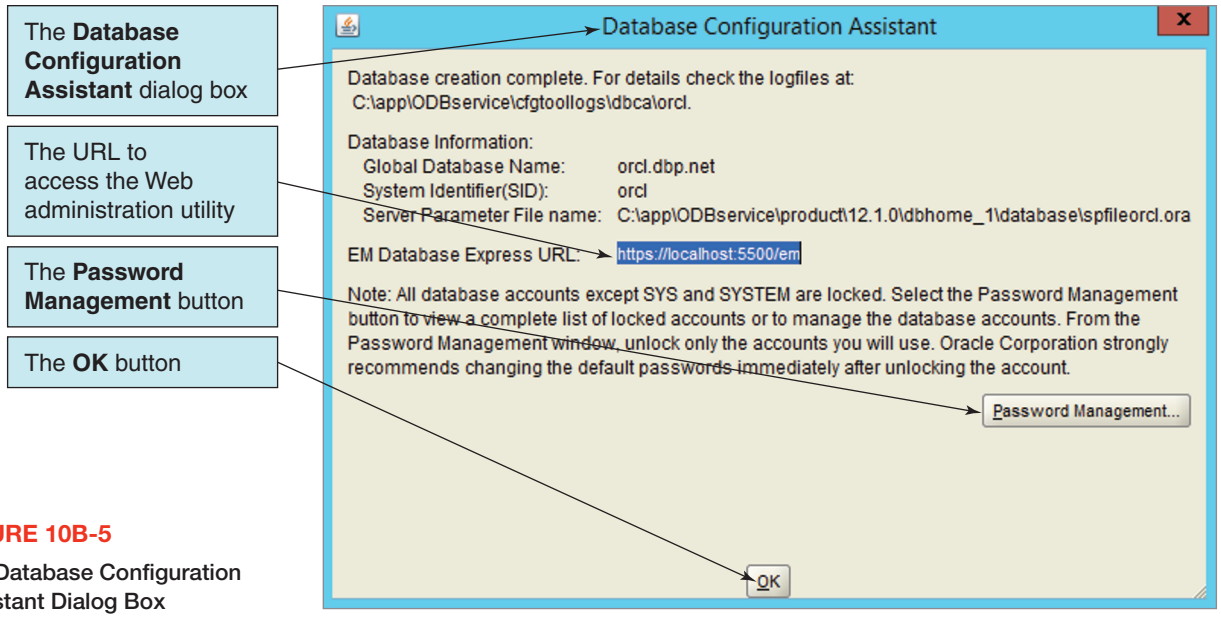


FIGURE 10B-5

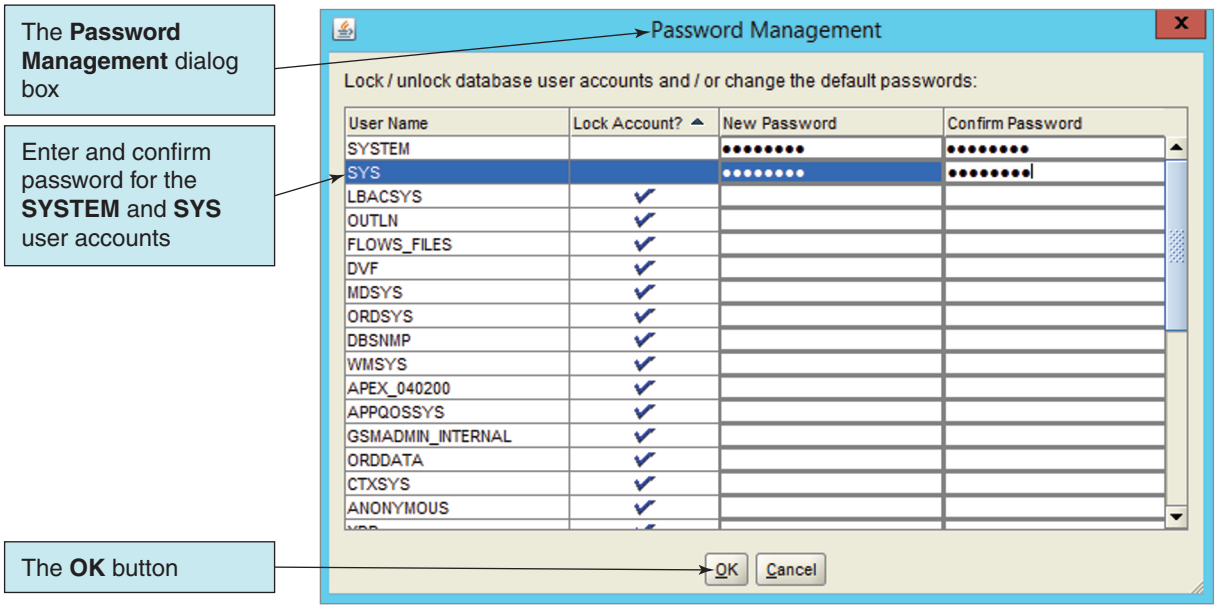
The Database Configuration Assistant Dialog Box

As part of the installation process, we specified that a new database instance named *orcl* should be created. The *orcl* database includes a sample database schema (we will discuss Oracle Database schemas later in this chapter) named *hr* that is used for examples in Oracle Database 12*c* documentation, and having this schema available will make it easier for you to use the documentation.

This is done by the Oracle Database Configuration Assistant, which is automatically run as part of the installation process and is shown in Figure 10B-5. At the end of this process, we are provided with a URL to use for Web administration of the database and given a chance to specifically set user account passwords. We use the Password Management dialog box, as shown in Figure 10B-6, to specifically set the SYSTEM and SYS user passwords. As we close these two dialog boxes, we see the Finish step of the installation process, as shown in Figure 10B-7. Click the Close button to complete the installation process.

FIGURE 10B-6

The Password Management Dialog Box



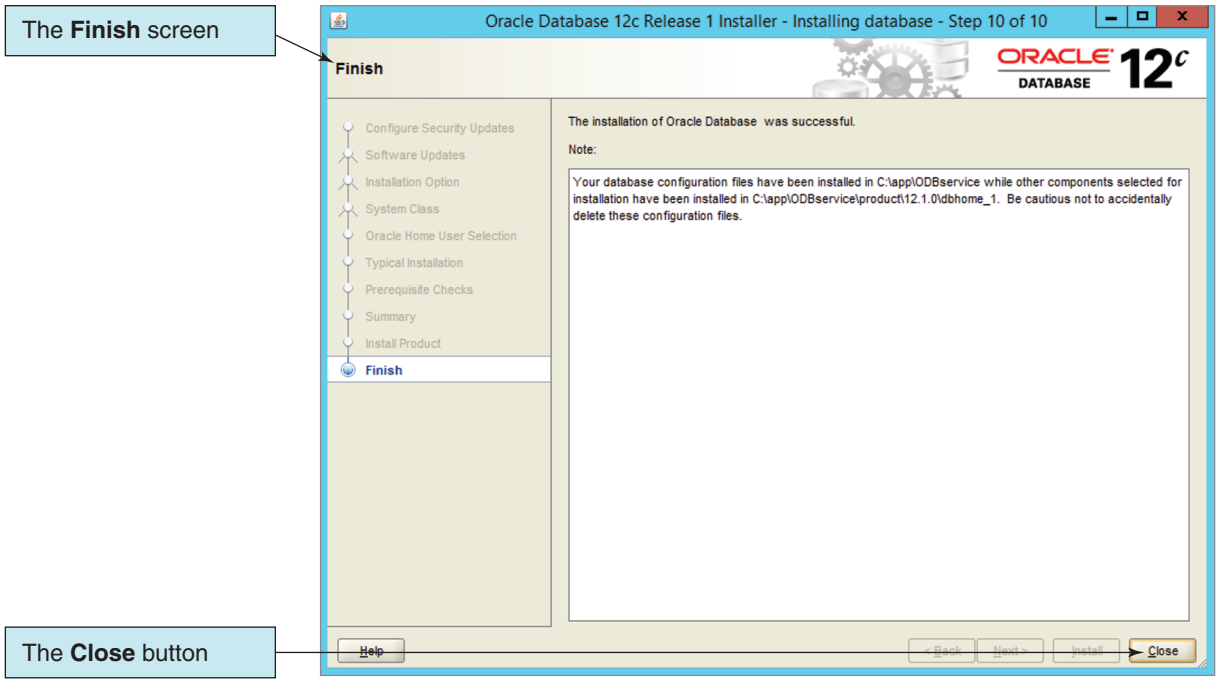


FIGURE 10B-7
The Finish Screen

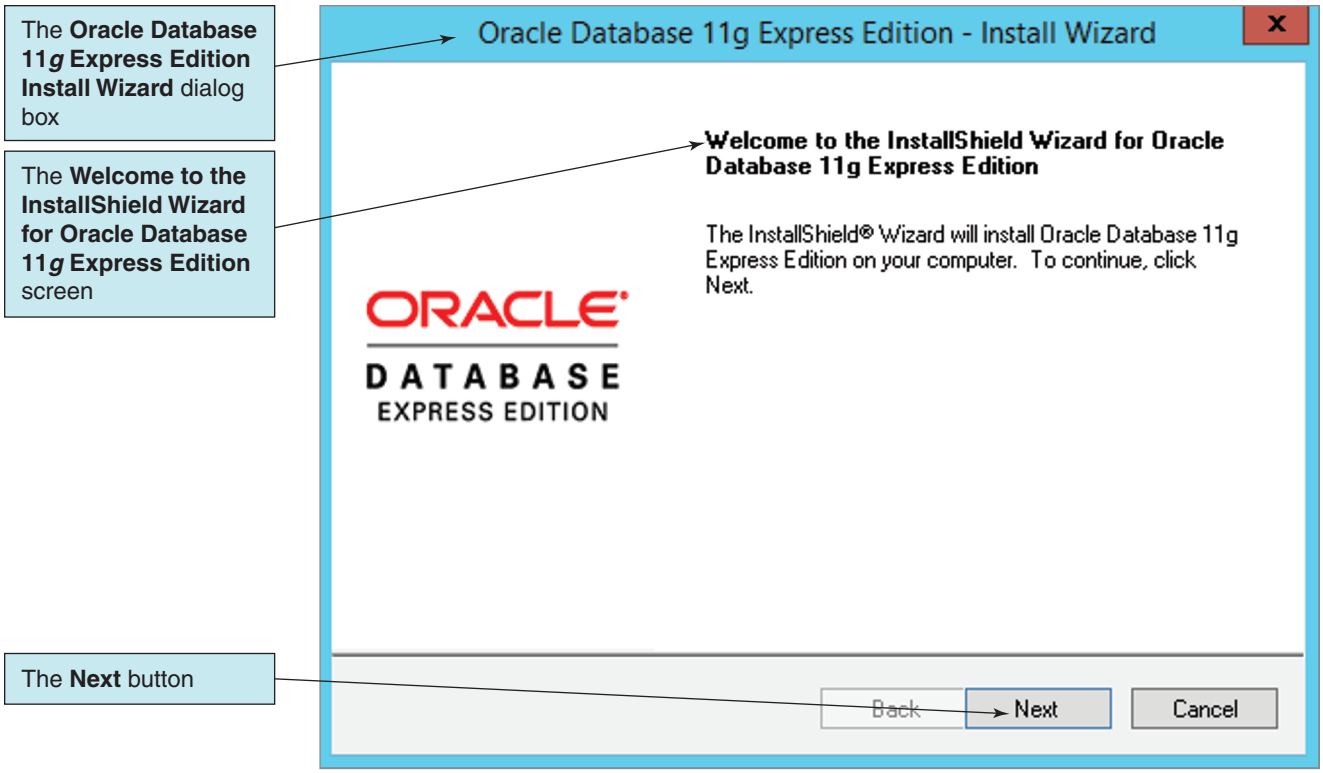
Installing Oracle Database Express Edition 11*g* Release 2 (Oracle Database XE)

Although we have discussed the installation of Oracle Database 12*c*, we do not recommend that you install this product yourself–let your DBA or your instructor do it. For personal use, we recommend using Oracle Database Express Edition 11*g* Release 2. In Oracle documentation, Oracle Database Express Edition 11*g* Release 2 is commonly referred to as **Oracle Database XE**, which is much easier to use, and we will use that name throughout this chapter. Although built on a slightly older version of Oracle Database, it will provide you with all the functionality you need to do the work in this book, and everything you learn will work in Oracle Database 12*c*. It is freely available for download at *www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html*.

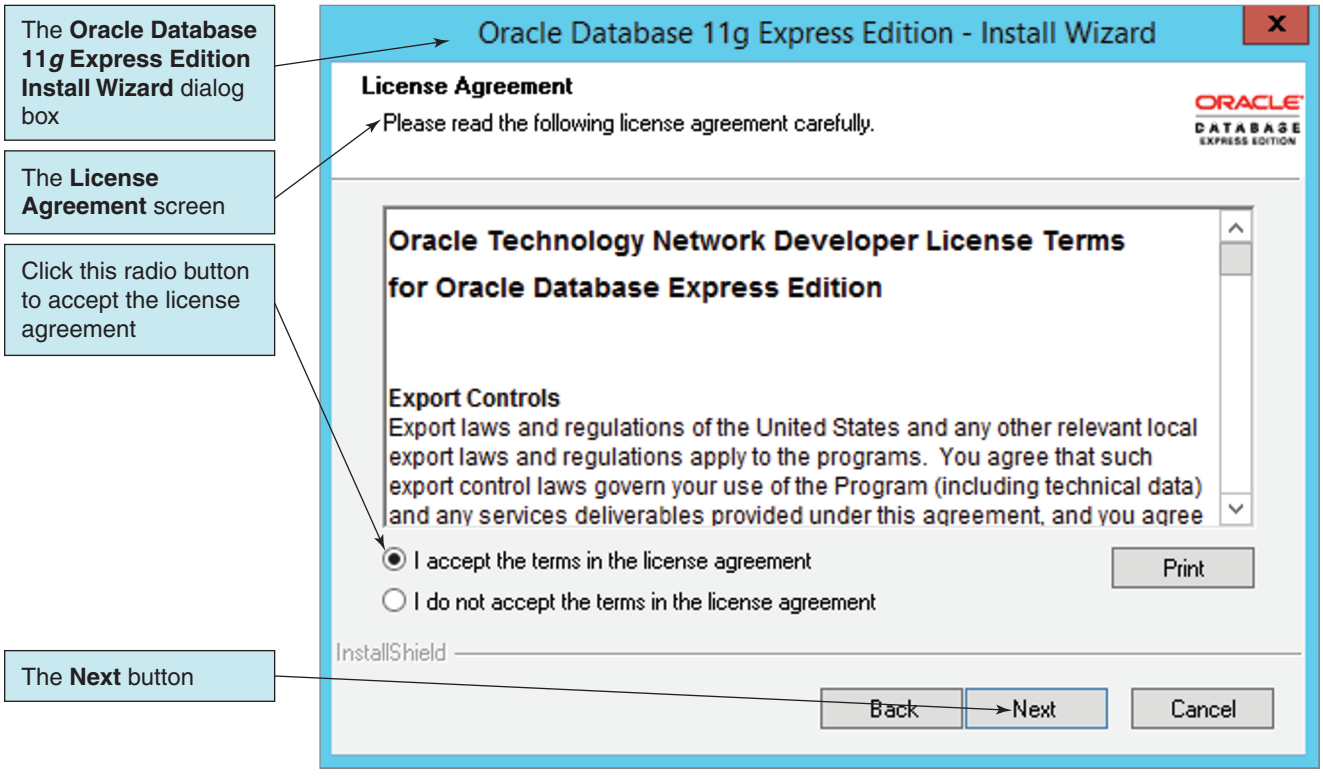
We will step through the installation process for Oracle Database XE on a Windows operating system.

Installing Oracle Database XE

1. Download the **OracleXE112_Win64.zip** file (or the **OracleXE112_Win32.zip** file if you have a 32-bit operating system) from *www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html*.
2. Extract the zipped file's contents to a folder–we created and used the folder **C:\Install\ODB_XE11R2**. The extraction will create a DISK1 folder (C:\Install\ODB_XE11R2\DISK1) and place all extracted folders and files in that folder.
3. To start the actual installation process, right-click the **setup.exe** file in the DISK1 folder, and click **Run as administrator**.
4. The **Oracle Database 11g Express Edition Install Wizard** is started and displays the **Welcome to the InstallShield Wizard for Oracle Database 11*g* Express Edition** screen, as shown in Figure 10B-8(a).
   - **NOTE:** The exact name that Oracle uses for Oracle Database Express Edition 11*g* Release 2 is inconsistent and varies from place to place. We will write the name that is currently being used when it is displayed in utilities such as the Install Wizard but will consider *Oracle Database Express Edition 11g Release 2* as the correct name and use it as the full product name. We will continue to use *Oracle Database XE* as our preferred name in this chapter unless a variant is indicated.



(a) The Welcome to the InstallShield Wizard for Oracle Database 11*g* Express Edition Screen



(b) The License Agreement Screen

FIGURE 10B-8

The Oracle Database 11*g* Express Edition Install Wizard

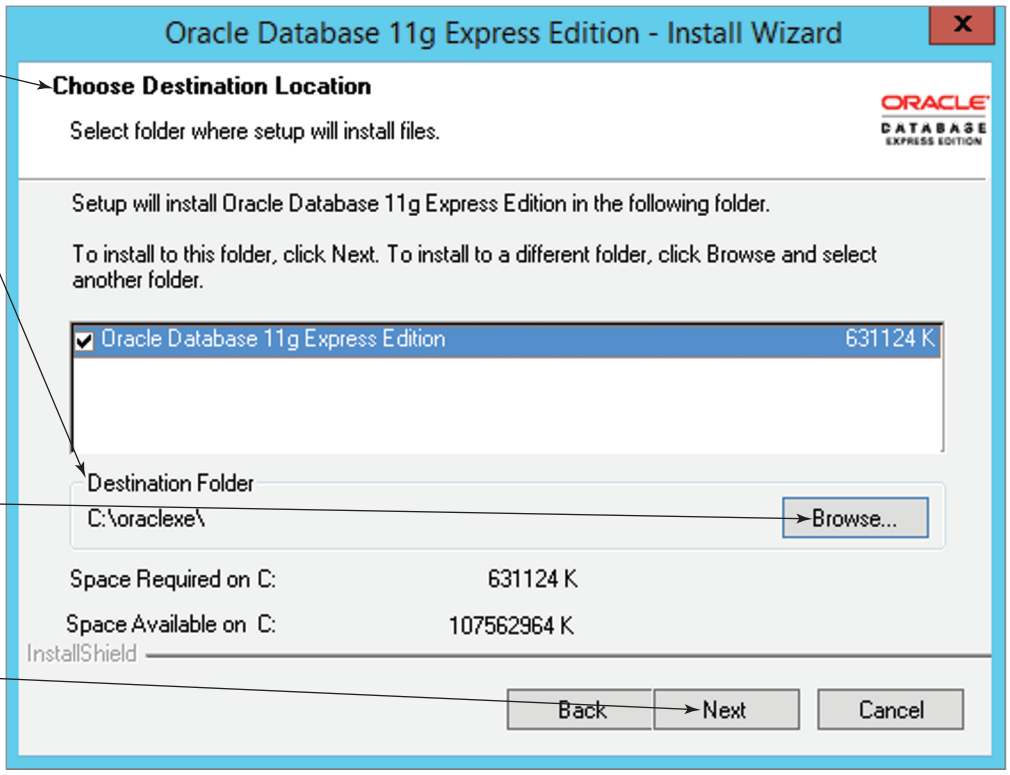
(*continued*)

The **Choose Destination Location** screen

The **Destination Folder** for the program installation

The **Browse** button

The **Next** button

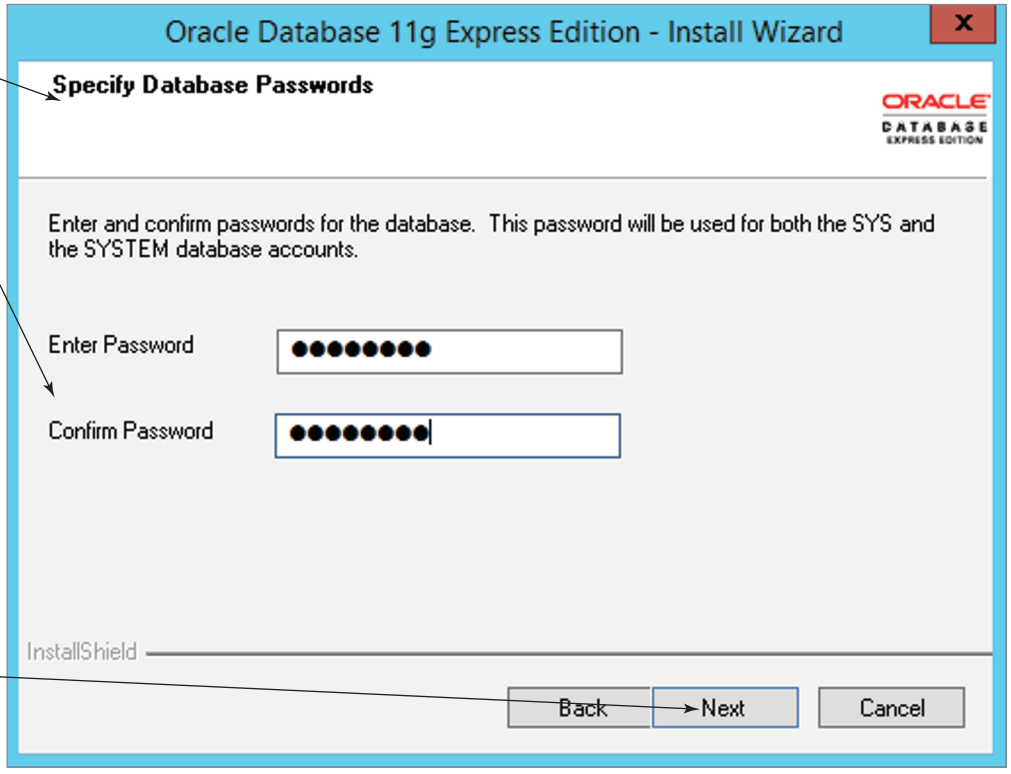


(c) The Choose Destination Location Screen

The **Specify Database Passwords** screen

Enter and confirm the password for the SYS and SYSTEM database user accounts

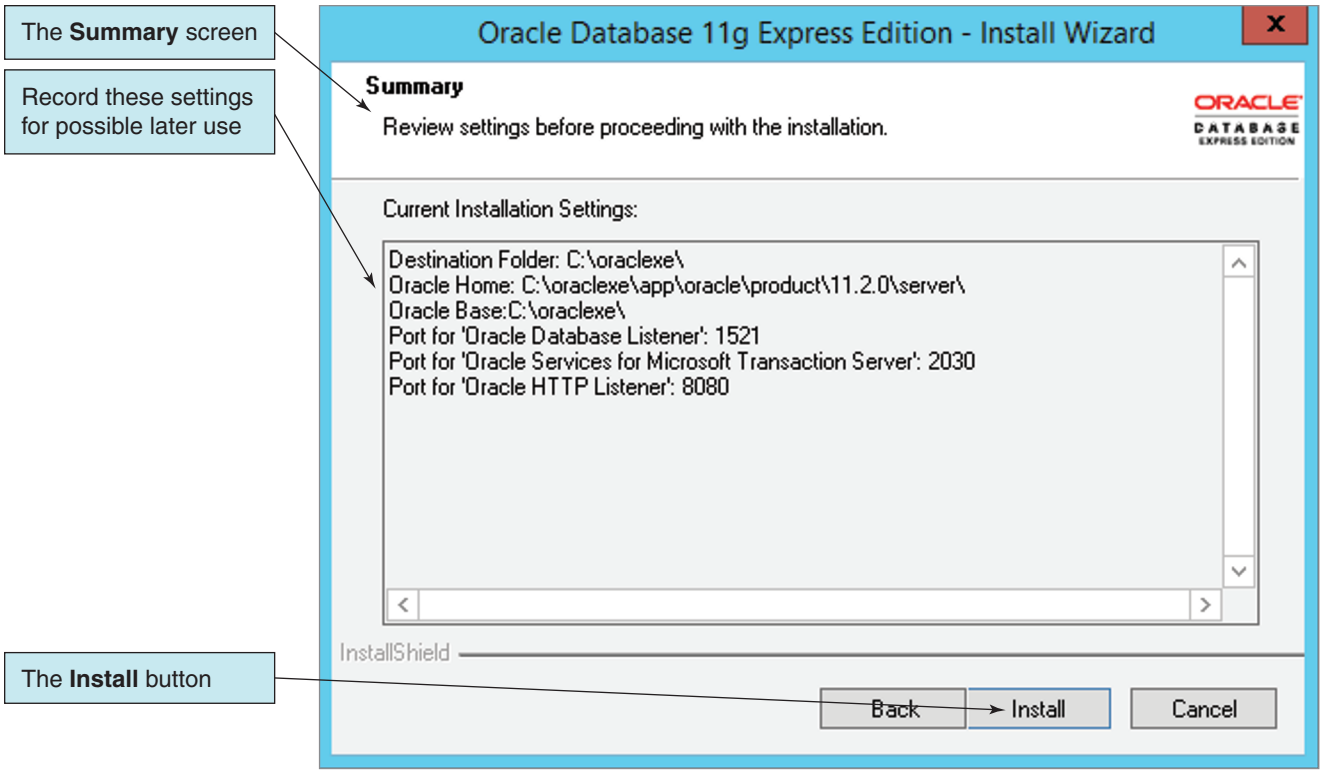
The **Next** button



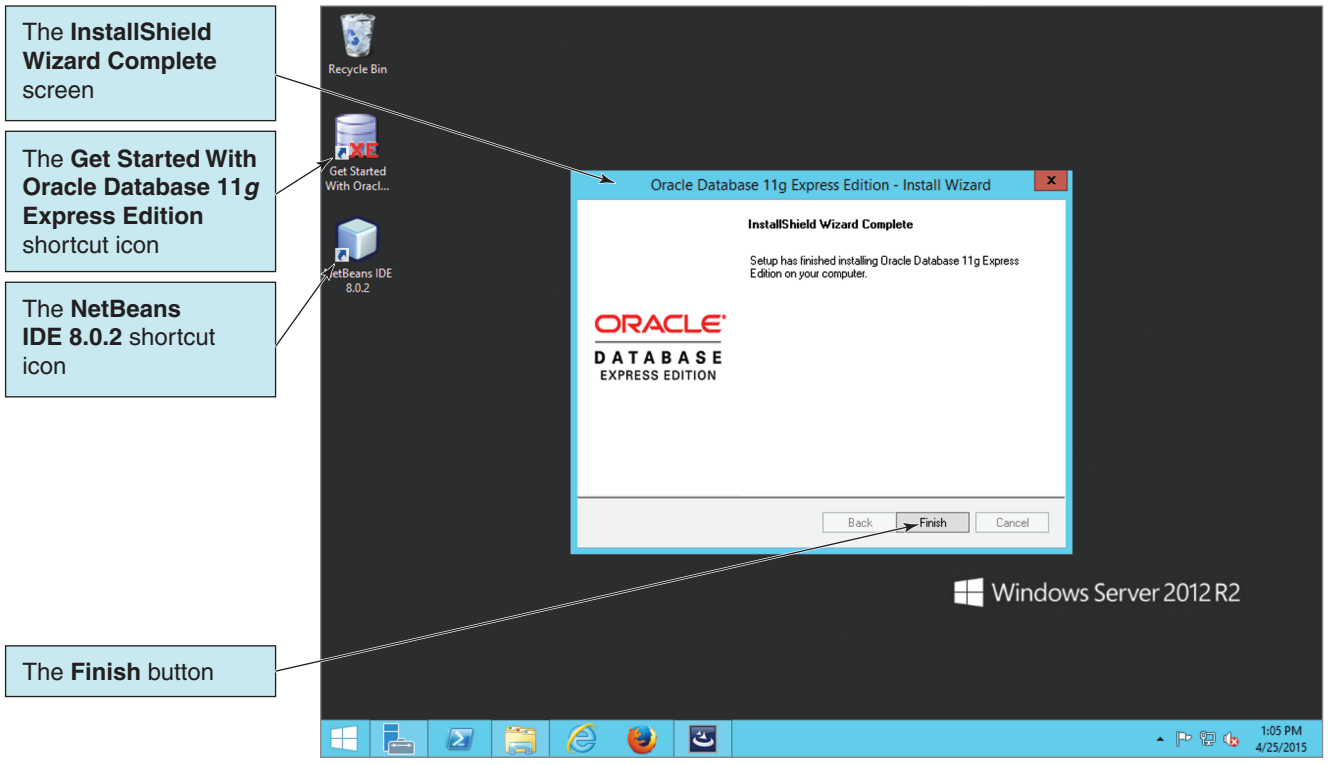
(d) The Specify Database Passwords Screen

FIGURE 10B-8

Continued



(e) The Summary Screen



(f) The InstallShield Wizard Complete Screen

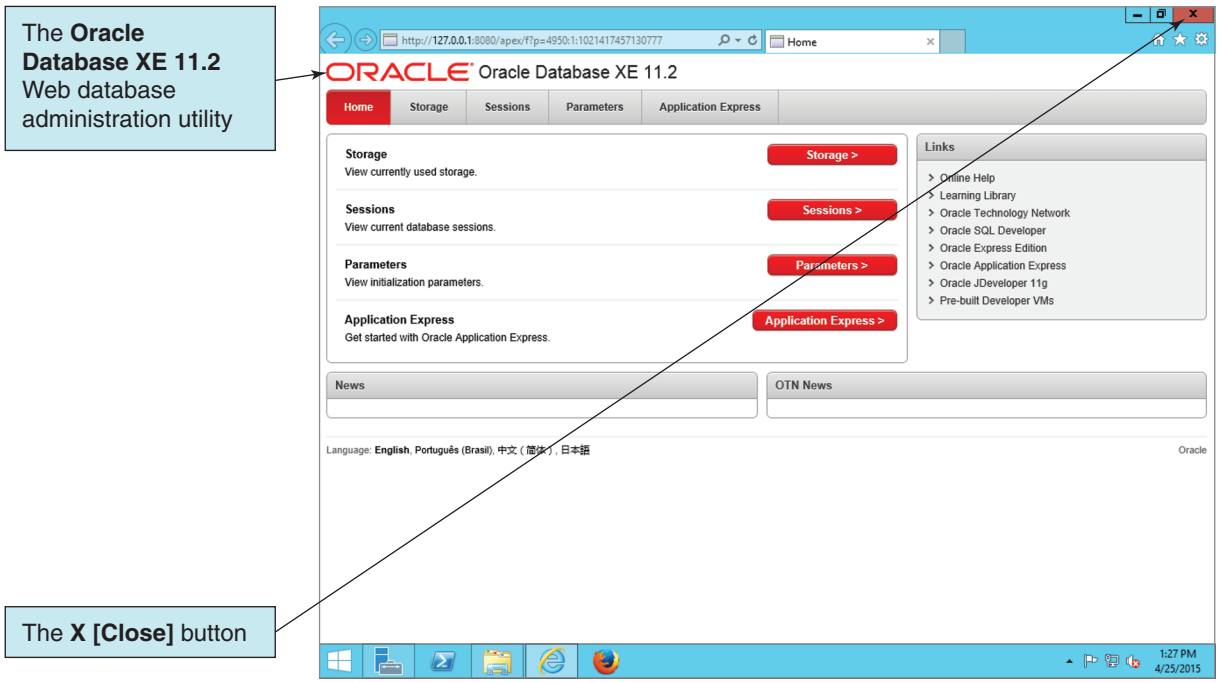
FIGURE 10B-8

Continued

5. Click the **Next** button. The **License Agreement** screen is displayed, as shown in Figure 10B-8(b).
6. Click the **I accept the terms in the license agreement** radio button to select it, and then click the **Next** button.
7. The **Choose Destination Location** screen is displayed, as shown in Figure 10B-8(c). The default location is C:\oraclexe, and we will use this location. If you want or need to install the program files in a different location, use the **Browse** button to select the alternate location.
8. Click the **Next** button. The **Specify Database Passwords** screen is displayed, as shown in Figure 10B-8(d). Enter the password that will be used for both the SYS and SYSTEM database user accounts.
9. Click the **Next** button. The **Summary** screen is displayed, as shown in Figure 10B-8(e). This screen lists installation settings and contains important data including file locations and TCP/IP port numbers. You should record this data for possible later use.
10. Click the **Install** button. The **Setup Status** screen is displayed during the installation and configuration of Oracle Database XE—this takes awhile, so be patient. When the installation is complete, the **InstallShield Wizard Complete** screen is displayed, as shown in Figure 10B-8(f). Note that the installation process has added a desktop shortcut icon labeled *Get Started With Oracle Database 11g Express Edition*. This shortcut is used to launch the Oracle Database Express Edition 11*g* Release 2 **Oracle Database XE 11.2** Web administration utility, which we will discuss and use later in this chapter.
    - **NOTE:** We have previously installed the Java JDK and NetBeans 8.0.2 as discussed in Appendix I. The NetBeans IDE 8.0.2 shortcut icon also appears on the desktop.
11. Click the **Finish** button to close the Oracle Database 11*g* Express Edition Install Wizard.
12. Double-click the *Get Started With Oracle Database 11g Express Edition* Desktop shortcut icon. The **Oracle Database XE 11.2** Web administration utility is launched and displayed, as shown in Figure 10B-9. We will use this Web utility later in this chapter for database administration.
13. Click the Web browser **X [Close]** button to close the Oracle Database XE 11.2 utility.

FIGURE 10B-9

The Oracle Database XE 11.2 Web Administration Utility



BY THE WAY After you have installed Oracle Database 12*c* or Oracle Database XE, you should check regularly for updated versions of the Oracle Database release you are using. These updates are used in lieu of service packs and patches to make sure your installation is as secure as possible.

Oracle Database Administration and Development Tools

We will use five administration and development tools with the Oracle Database DBMS:

- **Enterprise Manager 12*c* Database Express 12*c*—**the Oracle Database 12*c* Web-based administration tool
- **Oracle Database XE 11.2—**the Oracle Database Express Edition 11*g* Release 2 Web-based administration tool
- **Database Configuration Assistant (DBCA)—**an Oracle Database 12*c* GUI administration tool that is used to create, manage, and delete databases
- **SQL*Plus—**the classic Oracle Database command-prompt utility, used by both Oracle Database 12*c* and Oracle Database Express Edition 11*g* Release 2
- **SQL Developer—**the Oracle GUI development tool, used by both Oracle Database 12*c* and Oracle Database Express Edition 11*g* Release 2

Oracle Database separates database administration and database development functions among these tools (allowing for some overlap). We will discuss the administration tools first while we set up our database, followed by a discussion of the development tools as we build the database structures, populate the database, and create stored procedures and triggers.

While Oracle Database 12*c* and Oracle Database XE use different Web-based database administration tools, the SQL*Plus and SQL Developer database development tools work exactly the same regardless of which version of Oracle Database you are using. Therefore, we will do as much work in the SQL Developer GUI development tool as possible.

The Oracle Database 12*c* Configuration Assistant

Although an Oracle Database 12*c* database can be created via the **SQL CREATE DATABASE command**, the easiest way to do it is by using the **Oracle Database Configuration Assistant (DBCA)**, and this is the method you should use. Note that the DBCA was used during the installation process to create the *orcl* database.

In this chapter, we will use the Cape Codd database (and later the View Ridge Gallery database we designed in Chapters 5 and 6) for which we wrote the SQL statements in Chapter 2. At this point, we would expect to create a database named *Cape_Codd* for the Cape Codd Outdoor Sports database, and, in fact, we do so when working with the other DBMS products discussed in this book.

However, the term *database* in Oracle Database 12*c* refers to a **database instance**, which is the overall structure used by the DBMS to manage one or more of the sets of tables, views, and other related objects that we usually call a database. In Oracle Database 12*c*, we manage these sets of objects by creating a *tablespace* to hold them. Oracle Database allows us to create many tablespaces within one database instance, and because we have already created the *orcl* database instance, we can simply create a Cape Codd tablespace there. We will discuss tablespaces after we introduce the Oracle Enterprise Manager 12*c* Database Control utility in the next section.

If we do need to create another database instance, we would use the DBCA to do so. For example, we might have a very large database application that warrants its own database instance. To open the DBCA, use the Windows Server 2012 R2 **Database Configuration Assistant** app icon in the Apps window. The DCBA Welcome page is shown in Figure 10B-10.

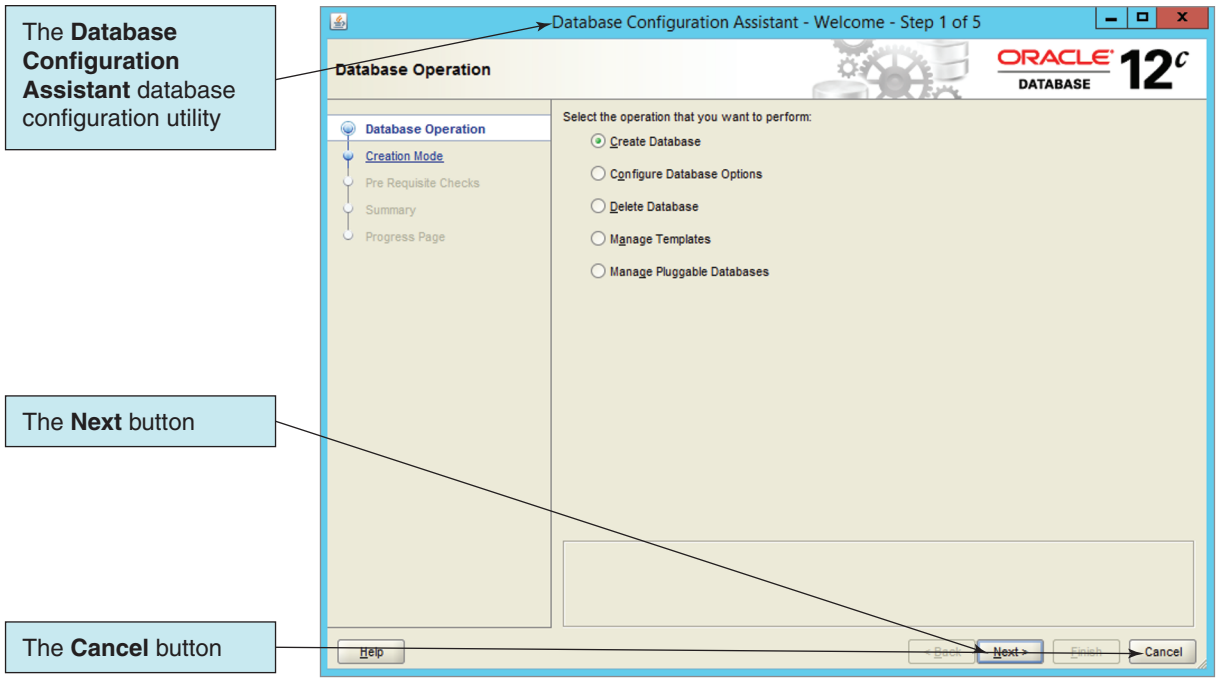


FIGURE 10B-10

The Oracle Database Configuration Assistant Utility

The Oracle Enterprise Manager Database Express 12c Database Administration Utility

Oracle Enterprise Manager Database Express 12*c* is a Web-based Oracle Database DBMS administration tool. The Oracle Database documentation refers to this tool as both the **Enterprise Manager** and as the **EM Express**. We will use the term *Enterprise Manager*, but remember that you will encounter both terms.

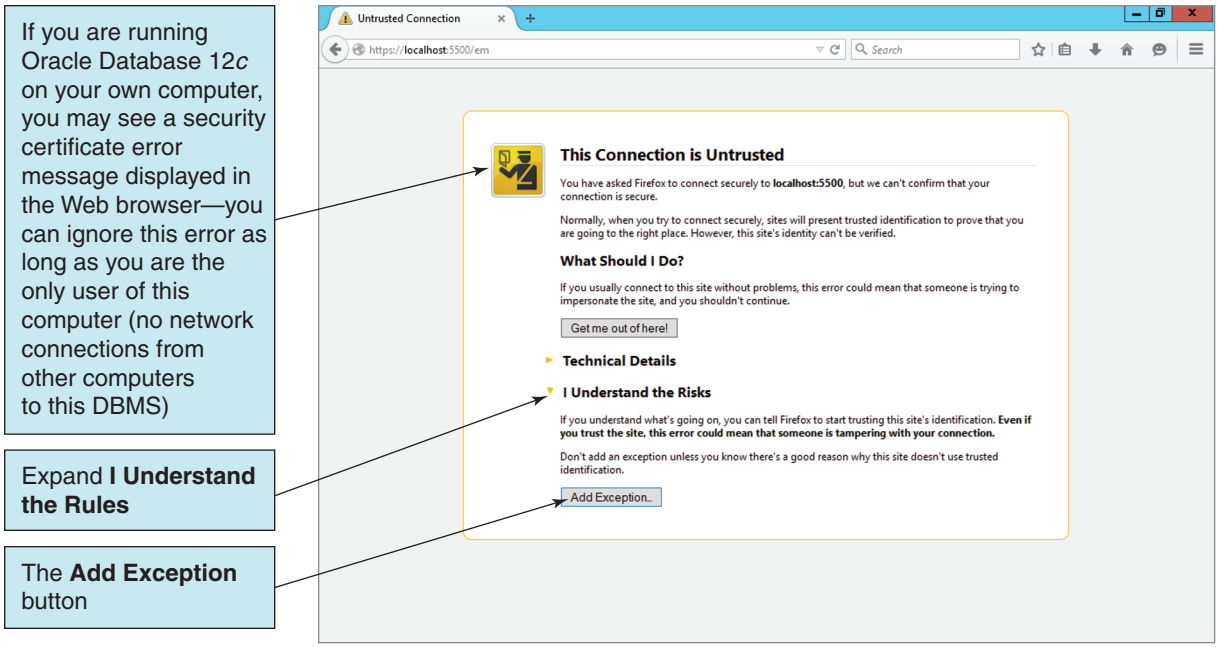
To access the Enterprise Manager, you open a Web browser and enter the URL ***https://{HostDNSName}:5500/em***, which in our case will be ***https://WS12R2-001.dbp.net:5500/em.*** If the Enterprise Manager is running on your own computer, you can also use the URL ***https://localhost:5500/em***.

Figure 10B-11(a) shows the Web page that appears in the Mozilla Firefox Web browser (each Web browser will display its own version of this warning) when there is a problem with the security certificate for this Web site, which will be displayed on your computer if there is not a properly installed security certificate for the Oracle Enterprise Manager Database Express 12*c* Web site. If you are installing the Personal Edition of Oracle Database 12*c* on your own computer, you will probably see this page. Security certificates and how to create and install them are topics beyond the scope of this book, but as long as this message is appearing on your own computer, you can ignore it and proceed to the Web site. Figure 10B-11(b) shows how to permanently store a security exception in Firefox so that you will not have to deal with this problem each time you open Enterprise Manager.

Figure 10B-12 shows the Enterprise Manager login screen. In Figure 10B-12, we are logging in as the **Oracle Database SYS system account** with the password that we assigned to this account during the installation process. When we use the SYS account, we connect as SYSDBA, as shown in the **Connect As** selection.

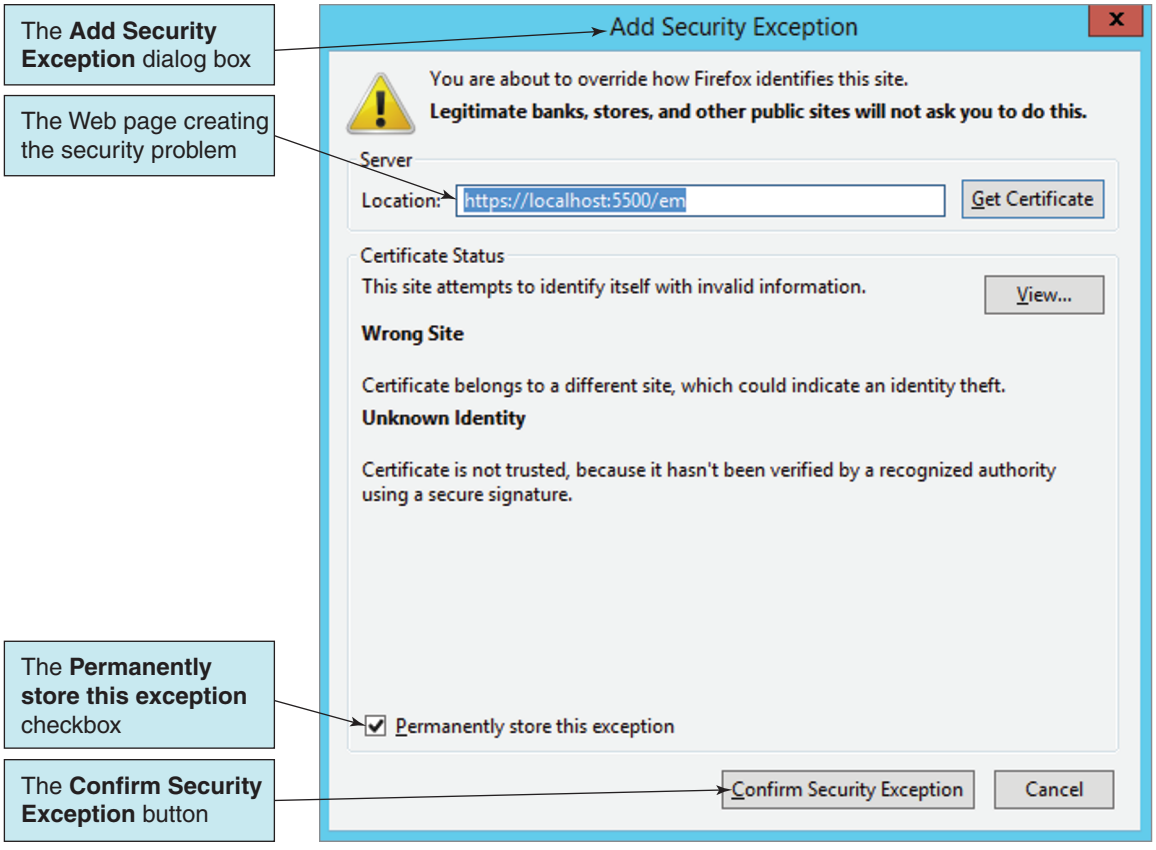
Figure 10B-13 shows Enterprise Manager displaying information about the *ORCL* database (*orcl* instance), shown on host server *WS12R2-001*—this is the database created during the installation process. The tabbed *ORCL (12.1.0.1.0)* (Database Home) page shown in Figure 10B-13(a) displays summary information about performance and other statistics about the database. If we click the *Performance* tab and then select *Performance Hub*, we will see the information and data shown in Figure 10B-13(b).

We will use the Enterprise Manager for DBMS administrative tasks in this chapter. Note that these screenshots show the *entire* Home and Performance Hub pages, but you will usually



(a) The Firefox Web Browser Untrusted Connection Page

(Firefox® is a registered trademark of the Mozilla Foundation, Mozilla Firefox, 39.0)



(b) The Firefox Web Browser Add Security Exception Dialog Box

(Firefox® is a registered trademark of the Mozilla Foundation, Mozilla Firefox, 39.0)

FIGURE 10B-11

The Web Site Security Certificate Problem Web Page

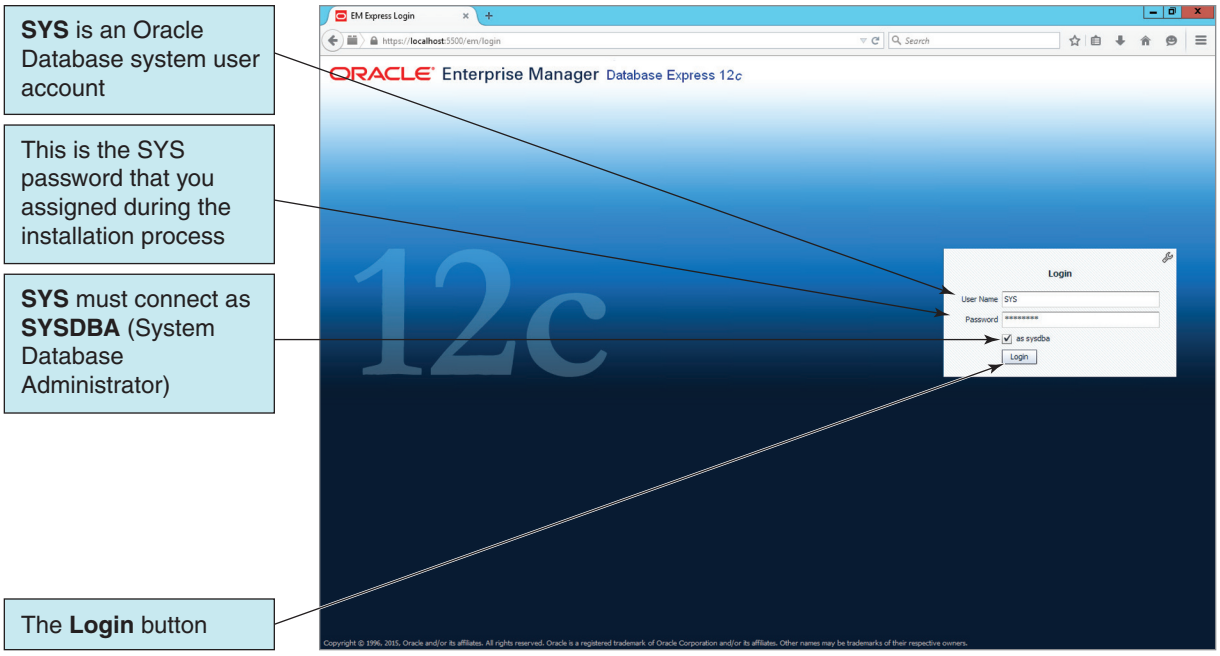


FIGURE 10B-12

The Oracle Enterprise Manager Database Express 12*c* Login Screen

(Firefox® is a registered trademark of the Mozilla Foundation, Mozilla Firefox, 39.0)

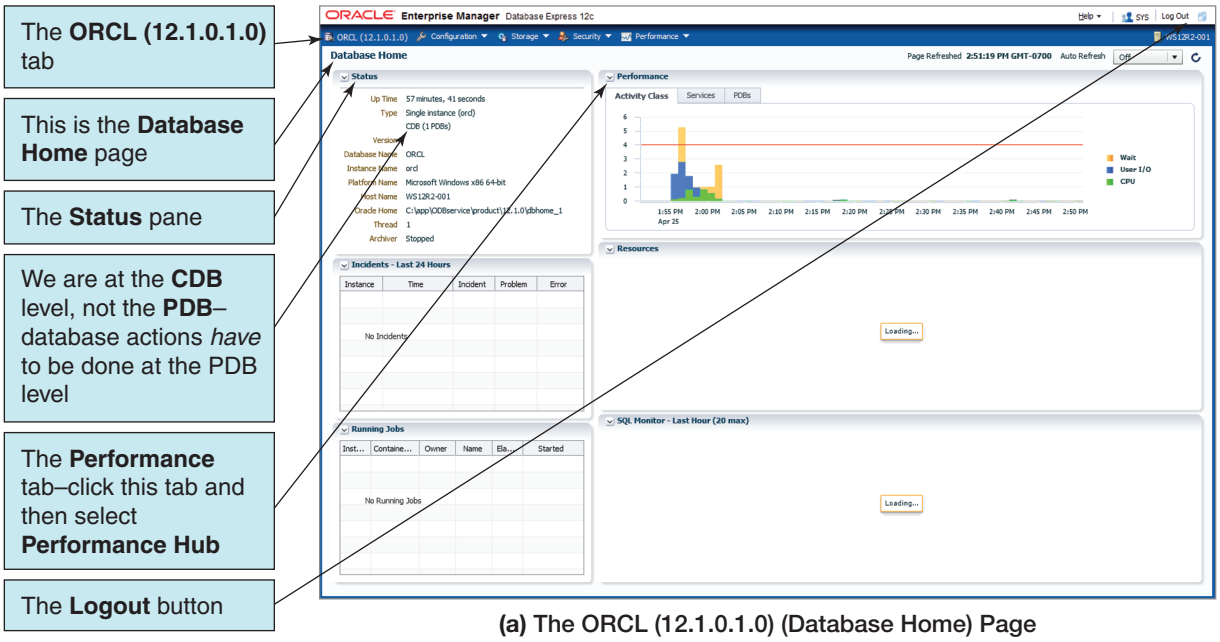
not be able to see all of these pages at the same time in a Web browser and will have to browse through the pages to see various parts of them.

Also note that Figure 10B-13(a) illustrates one of the new features in Oracle Database 12*c*: the separation of database administration into a parent **Container Database (CDB)** and a child **Pluggable Database (PDB)**. One CDB can work with (contain) one or more PDBs. This is intended to create database administration efficiencies by storing common Oracle Database objects in the CDB and Database-specific objects in each PDB.

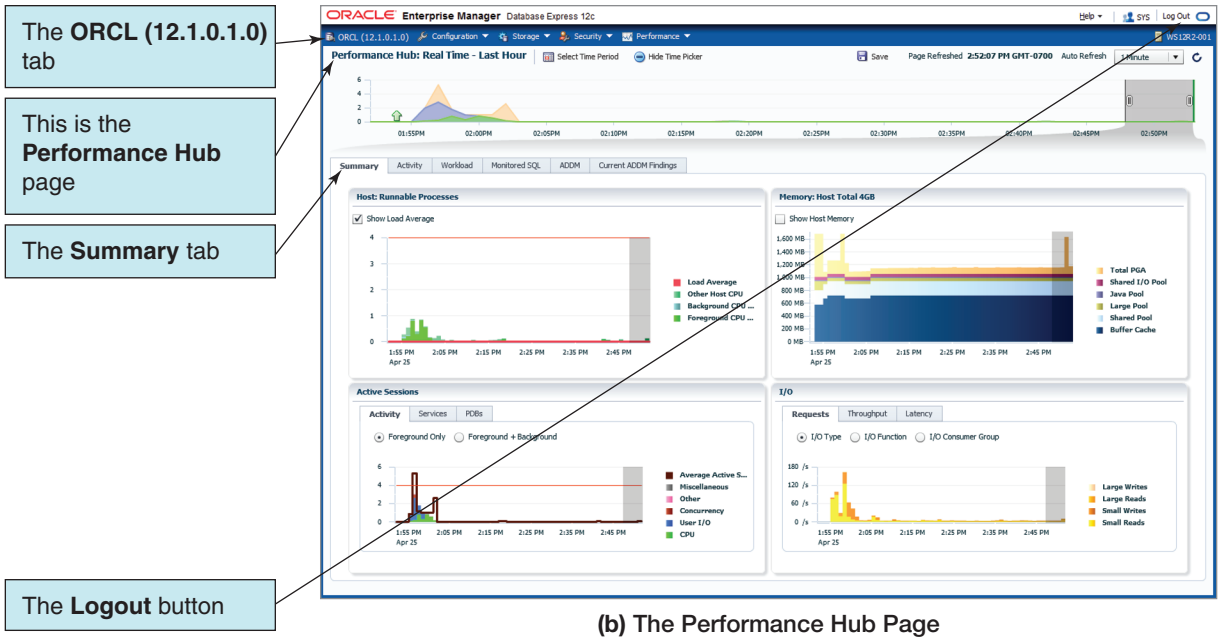
However, it also increases the complexity of Oracle Database 12*c* administration, and requires movement between the CDB and the PDB Enterprise Manager Web sites that was never required before. While we can create *common* users and roles at the CDB level (which can operate at the CDB and all subsidiary PDB levels), we *must* be at the *PDB* level to do database-specific work (such as creating database-specific users, roles, and tablespaces). In particular, tablespaces can only be created at the PDB level. Furthermore, setting up Oracle Database to allow using Enterprise Manager to switch between the CDB and the PDBs is an easy task, and requires use of the SQL*Plus command line utility. The detailed steps necessary to do this are beyond the

FIGURE 10B-13

The Oracle Enterprise Manager Database Express 12*c*



(a) The ORCL (12.1.0.1.0) (Database Home) Page



(b) The Performance Hub Page

FIGURE 10B-13
Continued

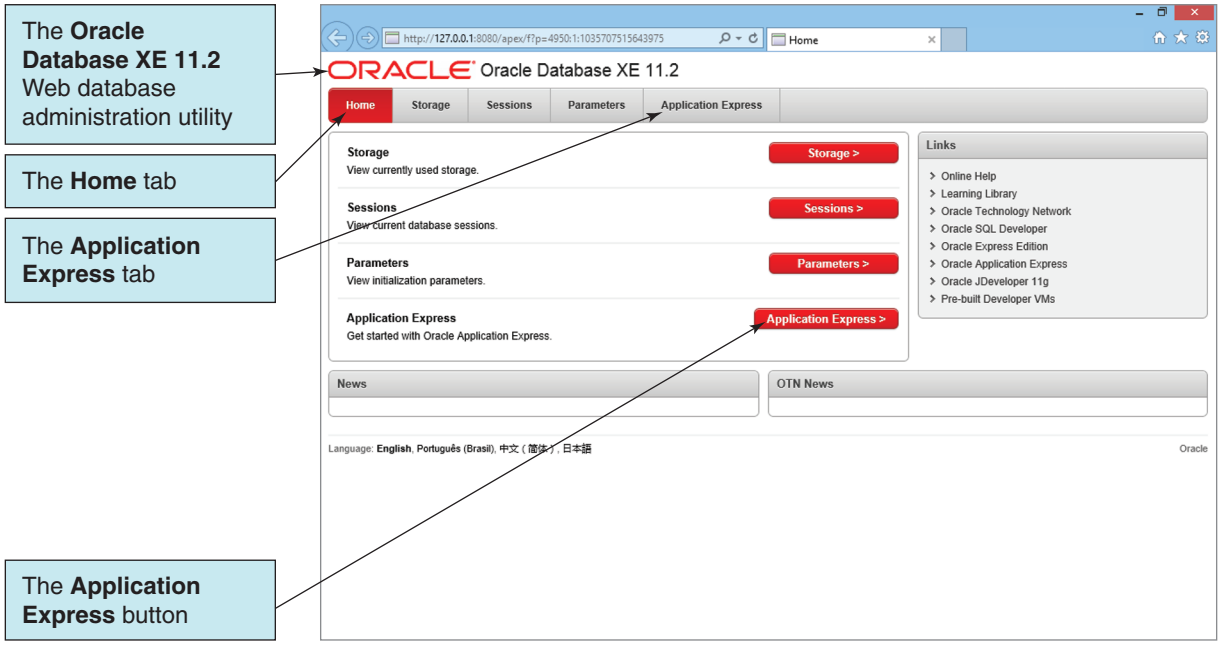
scope of this book. For these reasons, we recommend that, unless Oracle Database 12*c* has been installed for you by your instructor or workplace DBA, you download, install, and use Oracle Database XE. In this chapter we will illustrate and discuss a few of the administrative actions of Oracle Database 12*c*, but these are intended as introductory only, while we will cover Oracle Database XE in much more detail.

The Oracle Database XE 11.2 Database Administration Utility

Oracle Database XE 11.2 is a Web-based Oracle Database DBMS administration tool provided with Oracle Database XE. To access Oracle Database XE 11.2, double-click the desktop shortcut icon installed during the Oracle Database installation process. The Oracle Database XE 11.2 Home Page is shown in Figure 10B-14.

We create a new database in Oracle Database XE 11.2 by creating an **Oracle Application Express workspace**. We need to create the database for Cape Codd Outdoor Sports used in Chapter 2, so we will create the appropriate Application Express workspace.

FIGURE 10B-14
The Oracle Database XE 11.2 Home Page



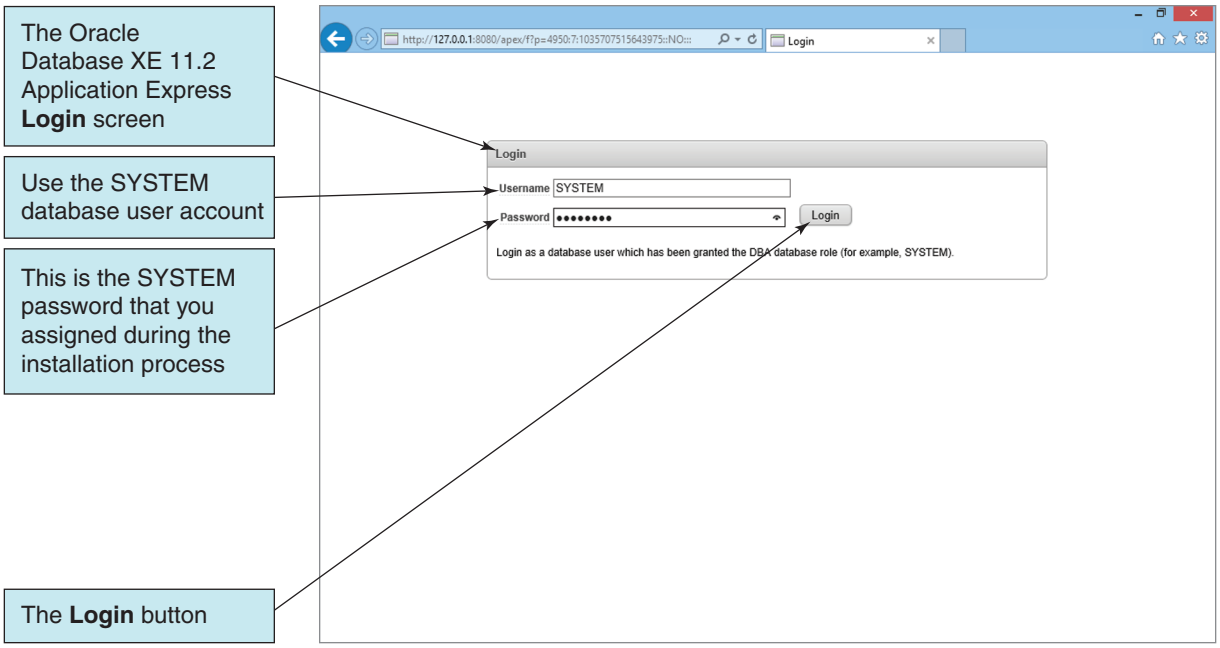


FIGURE 10B-15

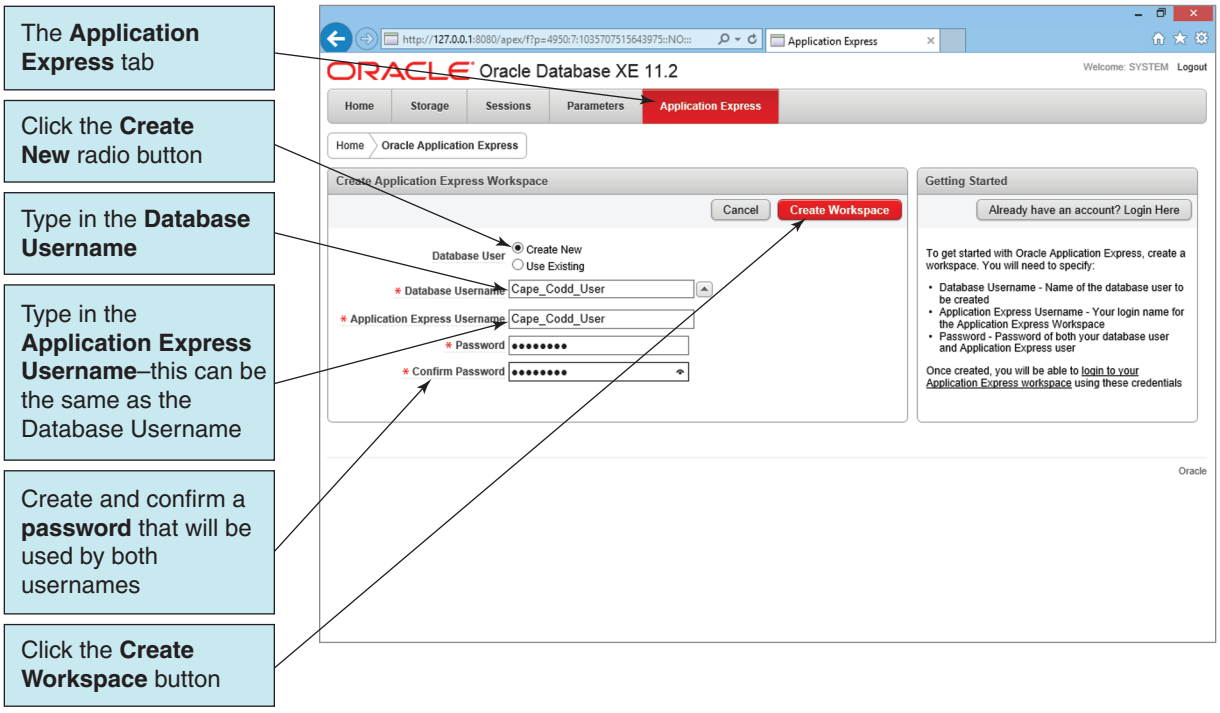
The Application Express Login Page

Creating an Oracle Application Express Workspace

1. On the Oracle Database XE 11.2 Home page, as shown in Figure 10B-14, click the Application Express button.
2. The Application Express **Login** page is displayed, as shown in Figure 10B-15.
3. Enter the username **SYSTEM** (which has DBA privileges) and the **password** you created for this account during installation. Click the **Login** button. The **Create Application Express Workspace** screen is displayed, as shown in Figure 10B-16.
4. Click the **Create New** database user radio button. Enter the new database username **Cape_Codd_User**, and use the same username as the application express

FIGURE 10B-16

The Create Application Express Workspace Page



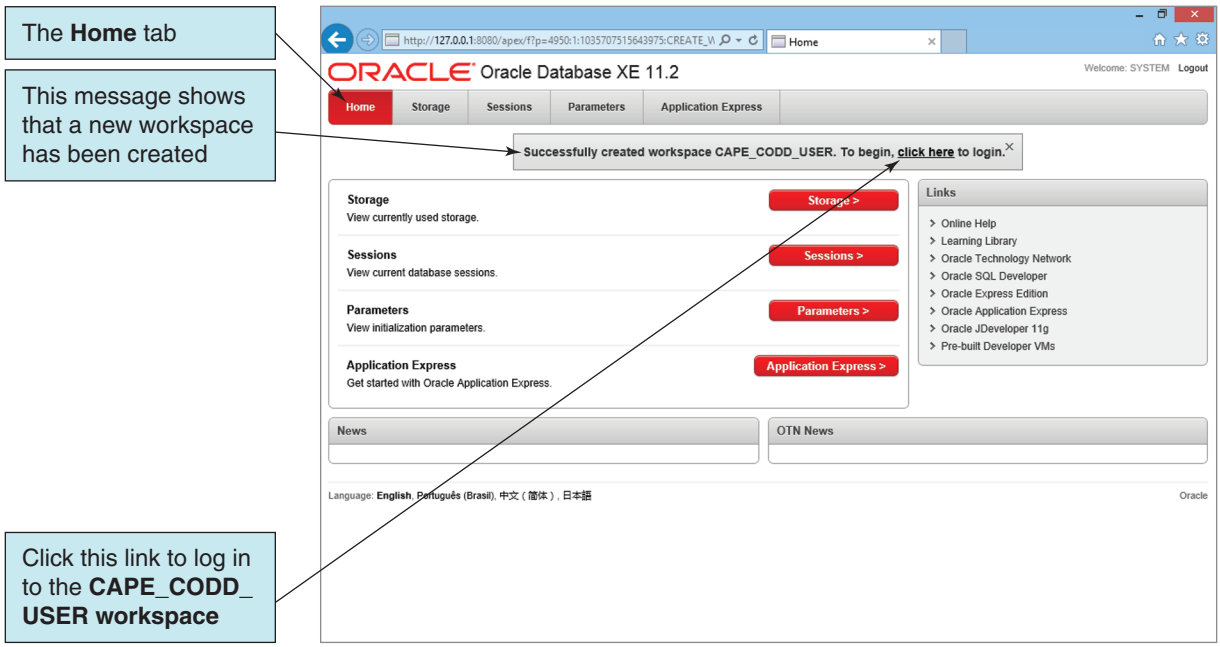


FIGURE 10B-17

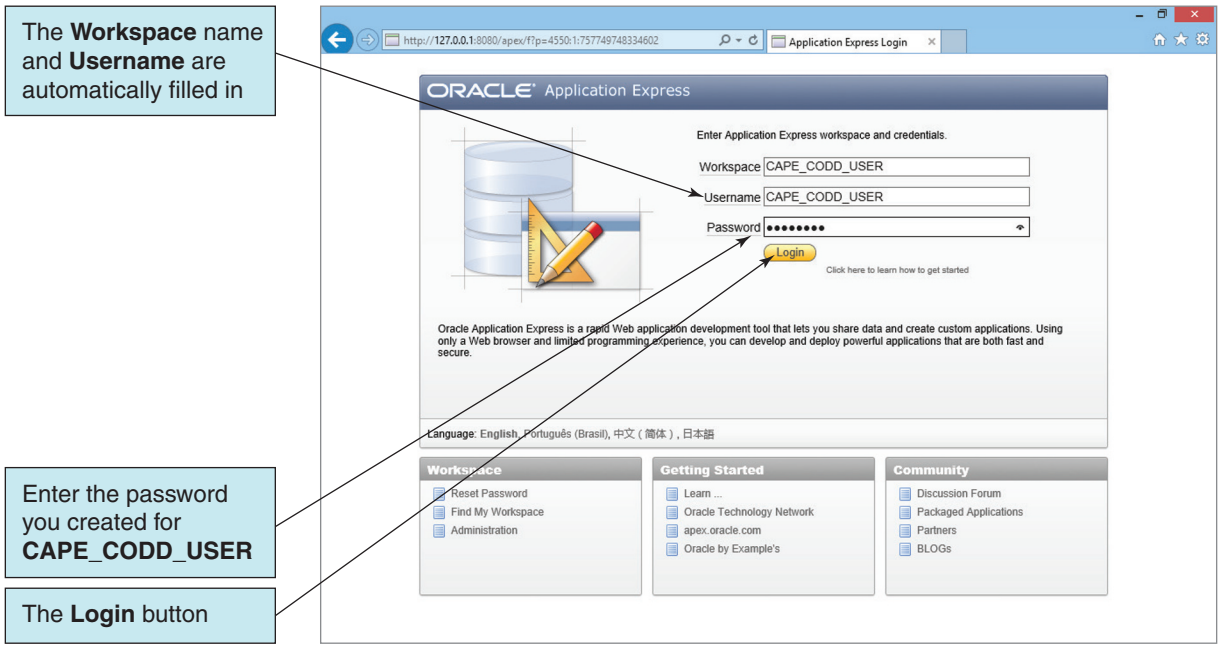
The Successfully Created Workspace Message

username. Create and confirm a **password** to be used by both the database user and the application express user.

5. Click the **Create Workspace** button. The Oracle Database XE 11.2 home page is displayed with the message *Successfully created workspace CAPE_CODD_USER. To begin click here to login*, as shown in Figure 10B-17. Note that Oracle Database has converted the username to all uppercase, which is normal behavior for Oracle Database.
6. Click the **click here** link. The **Oracle Application Express login page** is displayed, as shown in Figure 10B-18. Note that workspace and usernames are already provided. Enter the password you created for the CAPE_CODD_USER.
7. Click the **Login** button. The **Oracle Application Express Home page** for the CAPE_CODD_USER workspace is displayed, as shown in Figure 10B-19.

FIGURE 10B-18

The Application Express Workspace Login Page



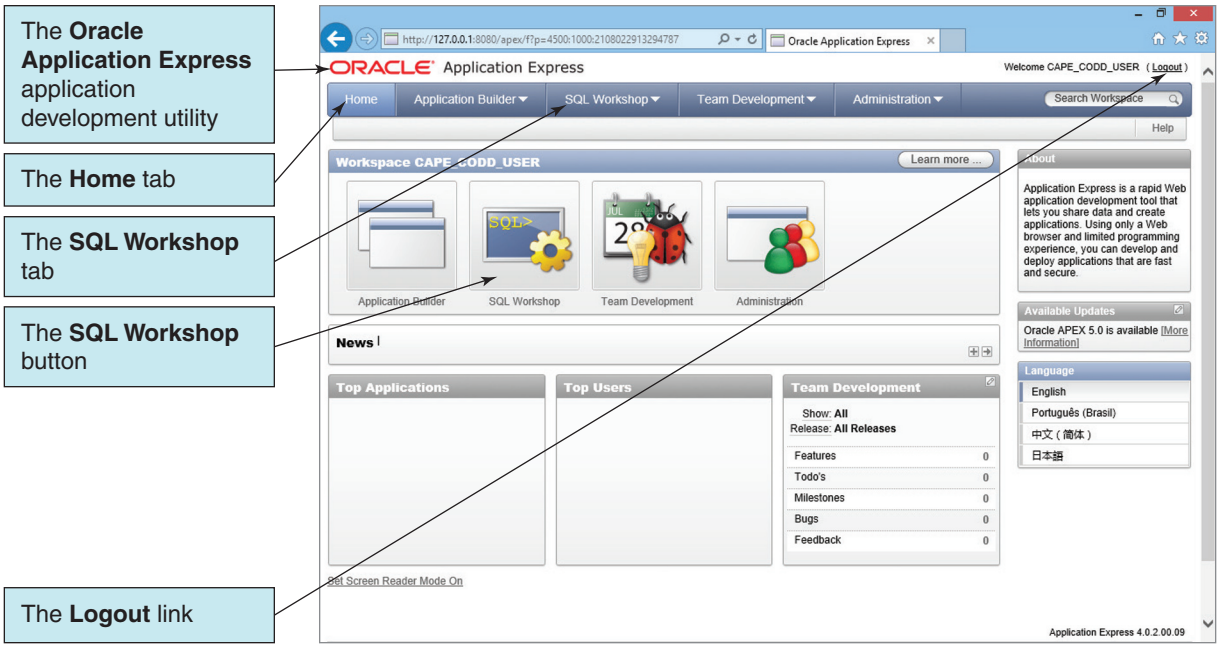


FIGURE 10B-19

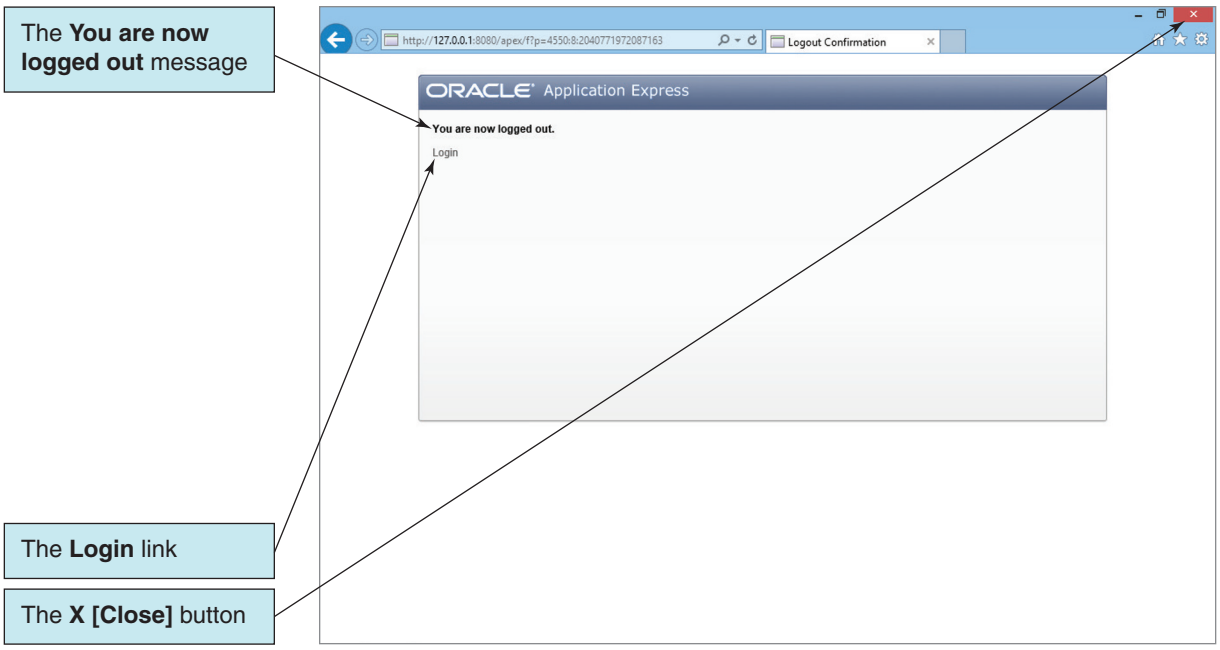
The Oracle Application Express Home Page

Although we won't use it in this book, Oracle Application is a Web-based application development environment that can be used for building Oracle Database databases and applications. Note, for example, the SQL Workshop tab, which provides an area for working with the SQL commands we discussed in Chapter 2 and Chapter 7. However, instead of using Oracle Application Express, we will use Oracle SQL Developer, discussed later in this chapter, as our database development tool.

8. Click the **Logout** link. The **Application Express "You Are Now Logged Out"** screen is displayed, as shown in Figure 10B-20. We have successfully created the workspace that will be the basis for our database for Cape Codd Outdoor Sports, so we are done using the Oracle Database XE 11.2 utility for now. Click the **X [Close]** button to close the Web browser.

FIGURE 10B-20

The Application Express "You Are Now Logged Out" Page



Oracle Database Tablespaces

An Oracle Database **tablespace** is a logical subdivision of an Oracle Database database instance that is used to group related tables, views, triggers, stored procedures, and similar objects. For example, we can create a CAPE_CODD tablespace in a PDB to hold the CAPE_CODD tables, views, and other objects for Cape Codd Outdoor Sports. Thus, an Oracle Database 12*c* tablespace can be used to correspond to what we would call the CAPE_CODD database in SQL Server 2014 or MySQL 5.6.

Each tablespace is associated with one or more **datafiles**, which provide the physical storage on a computer, and thus tablespaces provide a mapping to actual files located on the computer secondary memory (e.g., hard drives). Oracle Database also provides backup and recovery based on tablespaces.

When a database instance is created, Oracle Database automatically creates a default set of tablespaces, and some of these are illustrated in Figure 10B-21. The SYSTEM and SYSAUX tablespaces are used by Oracle Database for database management. The TEMP tablespace is used, as the name implies, for temporary storage, for example, during SQL statement processing.

The USERS tablespace is the default space for the nonsystem objects created by Oracle Database users.

- For *Oracle Database 12c*, we could keep the CAPE_CODD objects in the USERS tablespace, but a better practice is to create a separate tablespace for each database application.
- For *Oracle Database XE*, we do not need to create any tablespaces, and we do not need to assign tablespaces to users. That is all done automatically.

The rest of the discussion in this section applies to Oracle Database 12*c* PDB only and you may skip it if you are using Oracle Database XE.

Creating the Oracle Database 12c CAPE_CODD Tablespace

1. Open a Web browser, and set the browser to the a URL setup that connects to a PDB specific Enterprise Manager Login Web page (setting up this URL is beyond the scope of this book).
2. Log into Enterprise Manager as **SYS** connecting as **SYSDBA**.
3. Upon logging in, the ORCL(12.1.0.1.0) Database Home page is displayed.
4. Click the **Storage** tab to display the Storage menu, and then select Tablespaces.
5. Click the **Create** button. The Create Tablespace Wizard **General Page** is displayed.
6. In the **Name** text box, type in **CAPE_CODD**. Select **Smallfile** in the Bigfile section, and select **Online** in the Status section.
7. Click the right arrow button. The Create Tablespace Wizard **Add Datafiles Page** is displayed.
8. In the Datafiles text box, type CAPE_CODD. The Extent Management, Type, and State settings are correct.
9. The File Directory is correct, but change the File Size setting to **10 MB**.

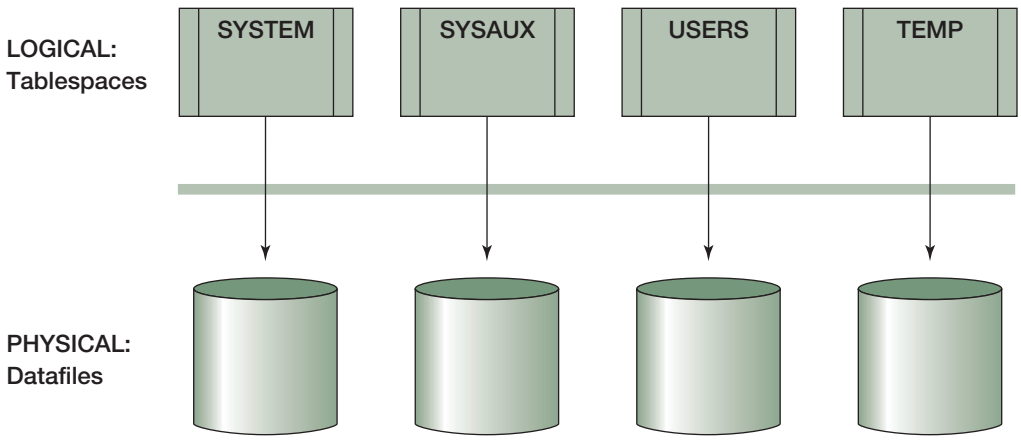


FIGURE 10B-21
The Oracle Tablespace and Datafile Structure

10. Check the **Automatically extend datafile when full (AUTOEXTEND)** check box, and set the Increment to 5 MB. Keep the unlimited Maximum File Size Setting.
11. Click the right arrow button. The Create Tablespace Wizard **Space Page** displayed.
12. Select **10MB** for the block size, and set Extent Allocation to **Automatic**.
13. Click the right arrow button. The Create Tablespace Wizard **Logging Page** displayed.
14. Select **Logging**, and then click the right arrow button. The Create Tablespace Wizard **Segments Page** displayed.
15. Select Automatic segment space management and None for compression.
16. Click the **OK** button to create the CAPE_CODD tablespace and associated datafile.
17. Click the **ORCL(12.1.0.1.0) tab** to return to the ORCL(12.1.0.1.0) Database Home page, then log out of the Enterprise Manager.

We have been logged in as SYS (connected as SYSDBA), and that is fine while we are performing database administration functions. But there will be problems if we try to build the CAPE_CODD database application logged in as SYS. The problems occur because system accounts such as SYS and SYSTEM primarily use the SYSTEM tablespace, and that is *not* where we want to place database application objects. Further, we *cannot* store triggers in the SYSTEM tablespace. So, now that we have created the place to build the CAPE_CODD database application, we need to create the Oracle Database user account that will be used to actually build it.

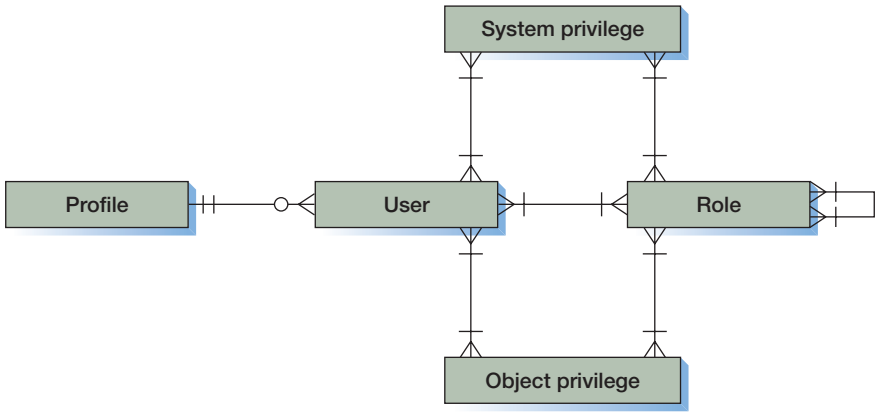
Oracle Database Security

Oracle Database provides robust and comprehensive security facilities. The relationships among the basic components of the Oracle Database security system are shown in Figure 10B-22. The components are User, Profile, Role, System privilege, and Object privilege. A User is a user account such as SYSTEM, Mary Jane, Fred, or some other user account.[8] A Profile is a set of system resource maximums that are assigned to an account. The Profile limits both computer and database resources; it is also used for password management, as described in the next section. As shown in Figure 10B-22, a User has exactly one Profile, but a Profile may be assigned to many accounts.

User Privileges

Each User can be allocated many System privileges, Object privileges, and Roles. A System privilege is the right to perform some action on the database data; on database structures, such as tables, views, or indexes; or on one of the Oracle Database system resources, such as a tablespace. One example is the CREATE VIEW system privilege. The interactions and

FIGURE 10B-22
Oracle Database Security Model



[8]Unfortunately, Oracle Database uses the word *SYSTEM* in two different ways here. There is an account named SYSTEM, and there are SYSTEM PRIVILEGEs.

restrictions of privileges in CDBs and PDBs is complex, and beyond the scope of the book (for a discussion, see *http://docs.oracle.com/database/121/CNCPT/cdblogic.htm#CIHFIAAA*).

An Object privilege is the right to perform an action (e.g., SELECT or UPDATE) on a specific object, such as the CUSTOMER table, the view CustomerInterestsView, and so on.

A Role can have many System privileges and/or object privileges, and it may also have a relationship to other Roles. The User account inherits the roles and privileges of each of the Roles it has been granted.

As shown in Figure 10B-22, a Role may itself have other Roles assigned to it. If so, it inherits the roles and privileges of those other Roles as well. Roles simplify the administration of the database. Without Roles, each account would need to be assigned the privileges that it needs, one by one. This time-consuming process would need to be repeated every time a new account is created. Using Roles, a set of privileges can be assigned to a Role just once—when a new User account is given that Role, all privileges of the Role are given to the new User account. Also, when a privilege is removed from a Role, all accounts that have that Role automatically have that privilege removed as well.

Creating a Common User Account

To create a new user account:

- For *Oracle Database 12c*, the Enterprise Manager is used to create a new user account.
- For *Oracle Database Express Edition 11g Release 2*, the new user account is created when we create a new Oracle Application Express workspace as discussed above.

We have already created the CAPE_CODD_USER user account in Oracle Application Express. Here we will create an Oracle Database 12*c* user named CAPE_CODD_USER (this is a variant of the username we use for SQL Server 2014 in Chapter 10A and for MySQL 5.6 in Chapter 10C, used here because Oracle Database, as we saw in Oracle Application Express, likes uppercase letters and underscores).

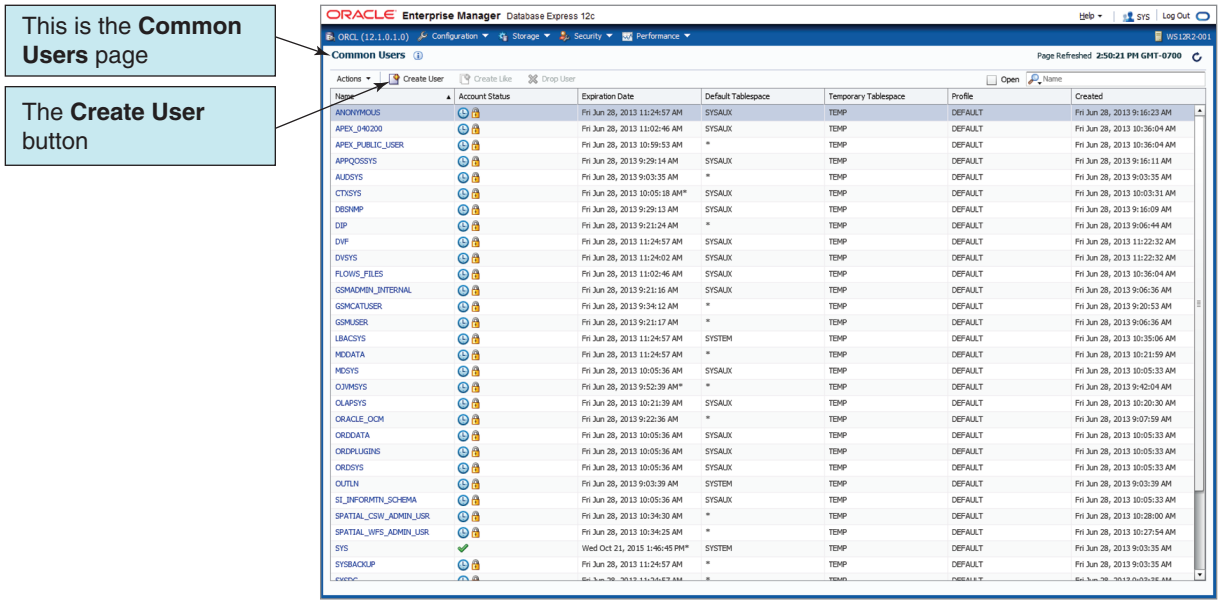
The rest of the discussion in this section applies to creating a common user in a CDB of Database 12*c* only, and you may skip it if you are using Oracle Database XE.

Creating the Oracle Database 12c CAPE_CODD_USER User Account

1. Log into Enterprise Manager as **SYS connecting as SYSDBA.** Use the password for SYS you created during the installation of Oracle Database 12*c*.
2. Starting at the ORCL(12.1.0.1.0) Database Home page, click the **Security** tab to display the Security menu, and then click **Users** to display the Common Users page as shown in Figure 10B-23.

FIGURE 10B-23

The Common Users Page



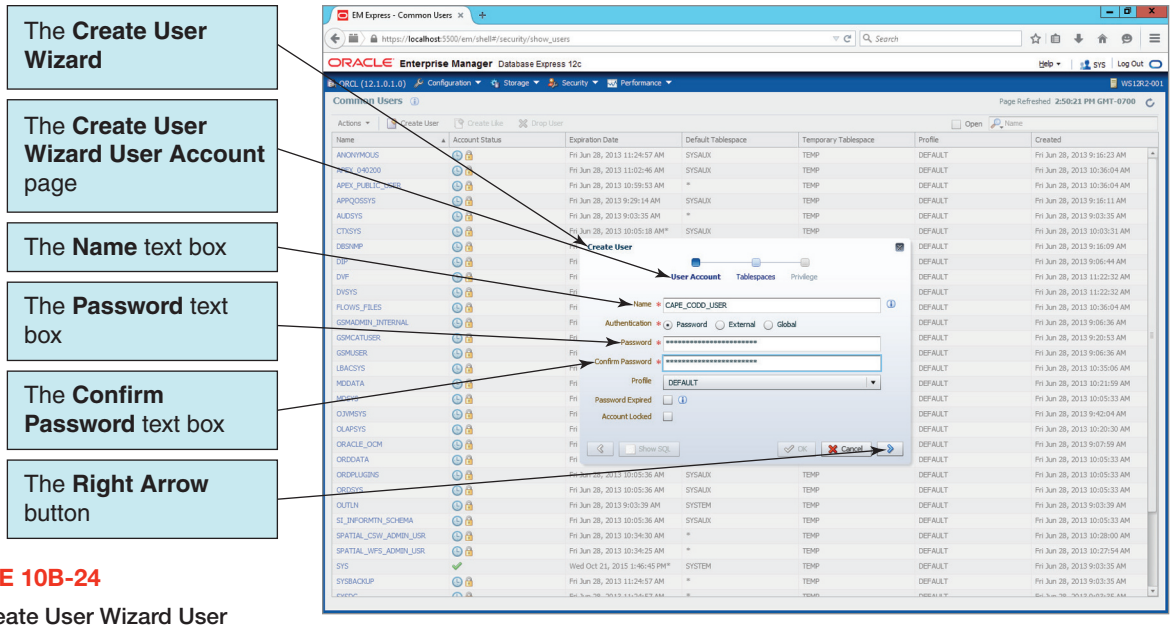


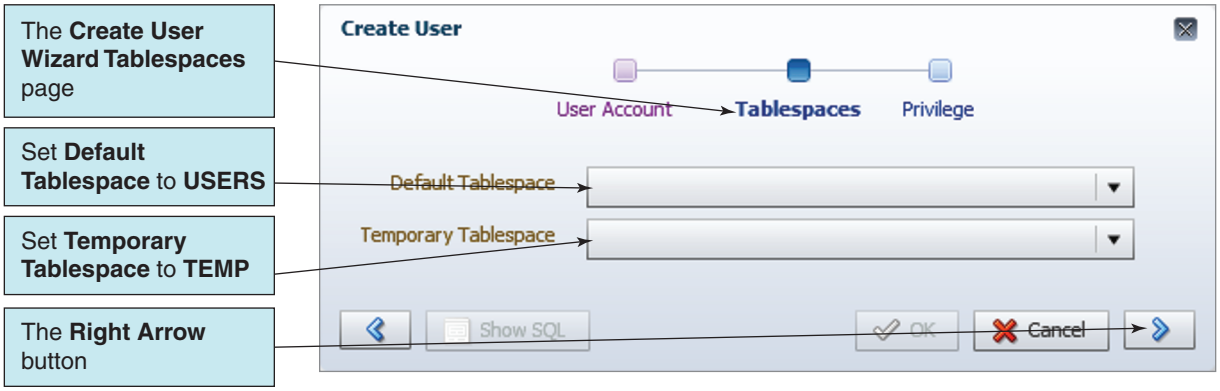
FIGURE 10B-24

The Create User Wizard User Account Page

3. On the Common Users page, click the **Create User** button to display the **Create User Wizard User Account** page, as shown in Figure 10B-24.
4. In the Name text box, type in **CAPE_CODD_USER**.
5. The Profile and Authentication settings are correct.
6. In the Enter Password and Confirm Password text boxes, enter the password **CAPE_CODD_USER+password**.
7. Click the **right arrow** button to display the **Create User Wizard Tablespaces** page, as shown in Figure 10B-25.
8. Set the Default Tablespace to **USERS**, and the Temporary Tablespace to **TEMP**, and then click the **right arrow** button to display the **Create User Wizard Privilege** page, as shown in Figure 10B-26.
9. Select the **RESOURCE** Role in the Name list, and click the right arrow button to assign it as a privilege to CAPE_CODD_USER, as shown in Figure 10B-27.
10. Select the **CONNECT** Role in the Name list, and click the right arrow button to assign it as a privilege to CAPE_CODD_USER.
11. Click the **OK** button to create CAPE_CODD_USER. A confirmation page is displayed.
12. Click the **ORCL(12.1.0.1.0) tab** to return to the ORCL(12.1.0.1.0) Database Home page, but stay logged into the Enterprise Manager.

FIGURE 10B-25

The Create User Wizard Tablespaces Page



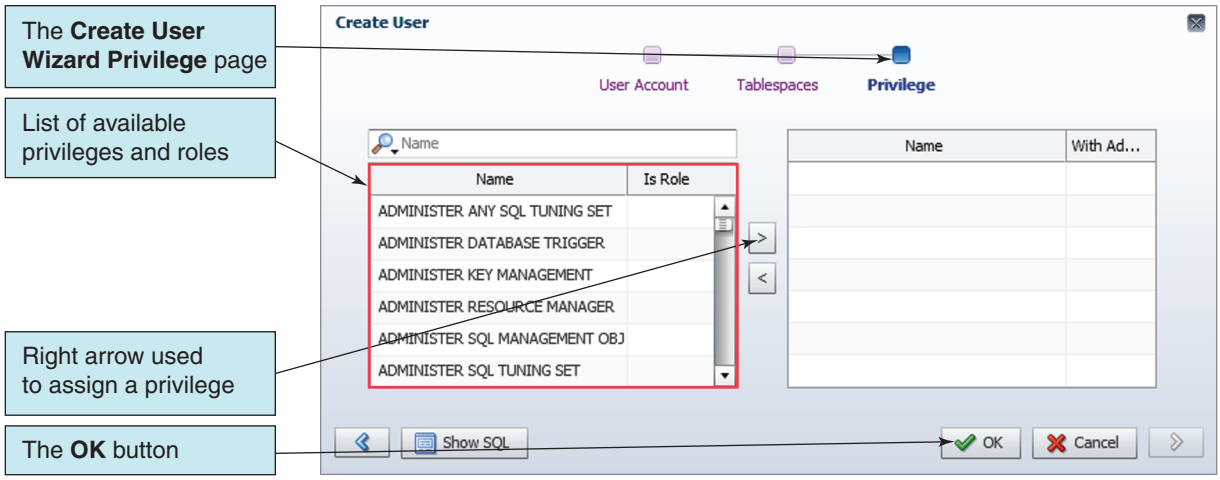


FIGURE 10B-26

The Create User Wizard Privilege Page

Creating a Common Role

When creating CAPE_CODD_USER for *Oracle Database 12c* in the Oracle Enterprise Manager, we assigned two predefined Roles to the User account: CONNECT and RESOURCE. CONNECT allows the user to connect to the database instance, and RESOURCE grants most of the System privileges needed to develop a database application. Therefore, these are typical Role assignments for a new user.

When creating CAPE_CODD_USER for *Oracle Database Express Edition 11g Release 2* in Oracle Database XE 11.2, we did not worry about assigning roles–the main users created in Oracle Application Express have sufficient database privileges that we do not need to worry about specific roles.

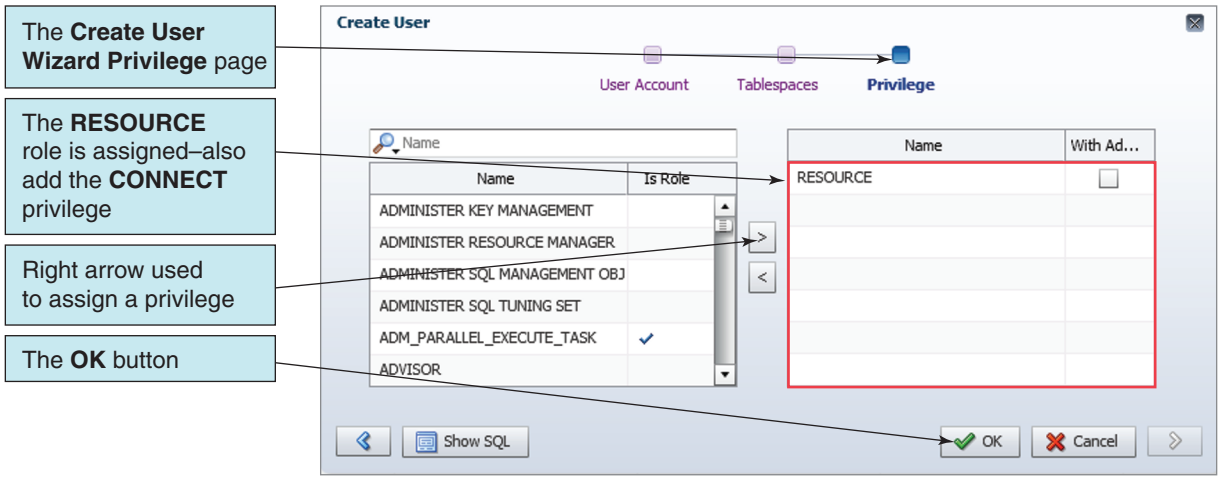
In Oracle Database 12*c*, however, the RESOURCE Role does not include the CREATE VIEW System privilege, which is needed to work with SQL views. Because CAPE_CODD_USER may need to create and use SQL views with the CAPE_CODD database application, we need to assign this System privilege to the user. The proper way to do this is to create a Role that is then granted the privilege and then assign that Role to the User.

Creating and Assigning an Oracle Database 12c Common Role

1. You should still be logged into Enterprise Manager as **SYS connecting as SYSDBA**. If not, log in now using the password for SYS you created during the installation of Oracle Database 12*c*.
2. Starting at the ORCL(12.1.0.1.0) Database Home page, click the **Security** tab to display the Security menu, and then click **Users** to display the Common Roles page as shown in Figure 10B-28.

FIGURE 10B-27

The Assigned RESOURCE Role



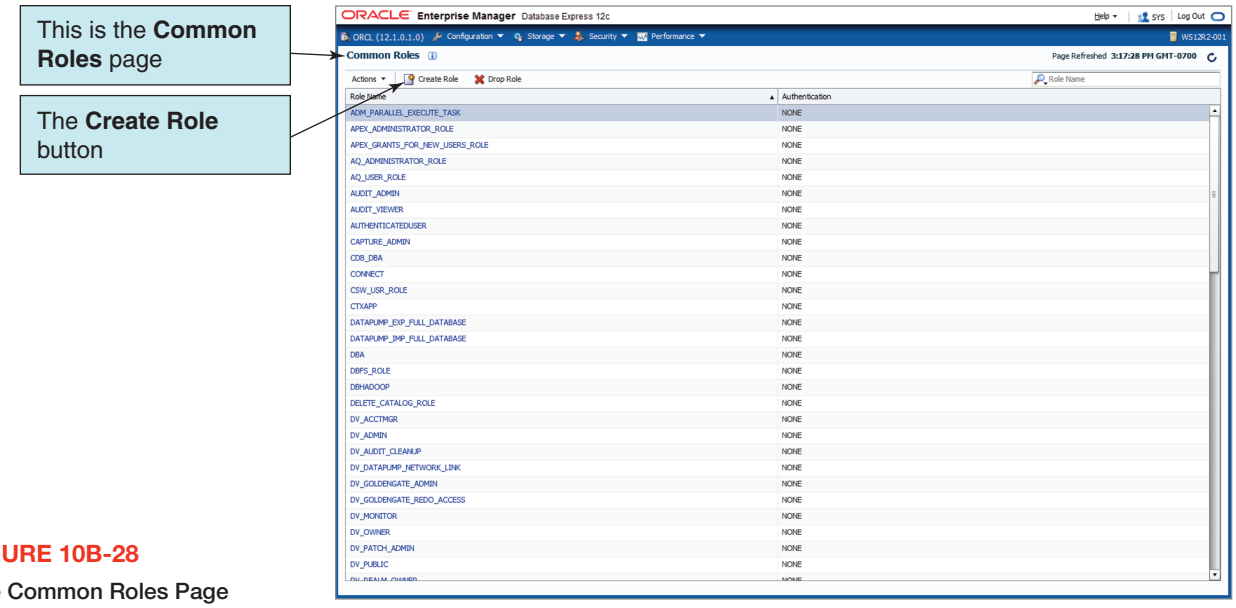


FIGURE 10B-28
The Common Roles Page

3. On the Common Roles page, click the **Create Role** button to display the **Create Role Wizard New Role** page. In the Name text box, type in **CAPE_CODD_DEV** as shown in Figure 10B-29.
4. Click the right arrow button to display the **Create Role Wizard Privilege** page. Select the **CREATE VIEW** Role in the Name list, and click the right arrow button to assign it as a privilege to CAPE_CODD_DEV, as shown in Figure 10B-30.
5. Click the **OK** button to create the CAPE_CODD_DEV role. A confirmation page is displayed.
6. Click the **ORCL(12.1.0.1.0) tab** to return to the ORCL(12.1.0.1.0) Database Home page, and then log out of the Enterprise Manager.

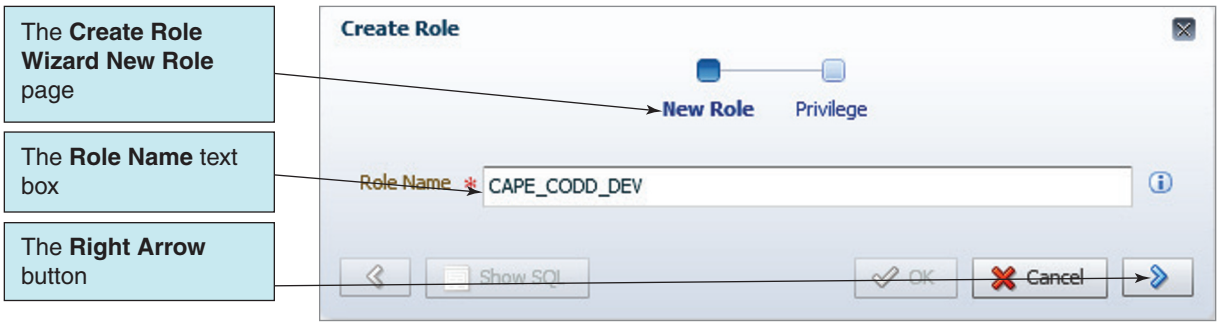
Now that we have created the CAPE_CODD_DEV Role, we need to assign it to CAPE_CODD_USER. This is simply a variation of the steps we used to create the CAPE_CODD_USER account, using the **Common Users |Actions | Alter Account** command, and you should be able to accomplish this task.

We have now completed the database administration steps we need to take in Enterprise Manager and Oracle Database XE 11.2 before we can actually start building the CAPE_CODD database application. We will build it in Oracle Database XE.

Oracle Database Application Development Tools

Now that we have completed our review of the three Oracle Database database administration tools, we can look at the two Oracle Database tools used for application development (as mentioned earlier, Oracle Application Express can also be used for application development, but that topic is beyond the scope of this book).

FIGURE 10B-29
The Create Role Wizard New Role Page



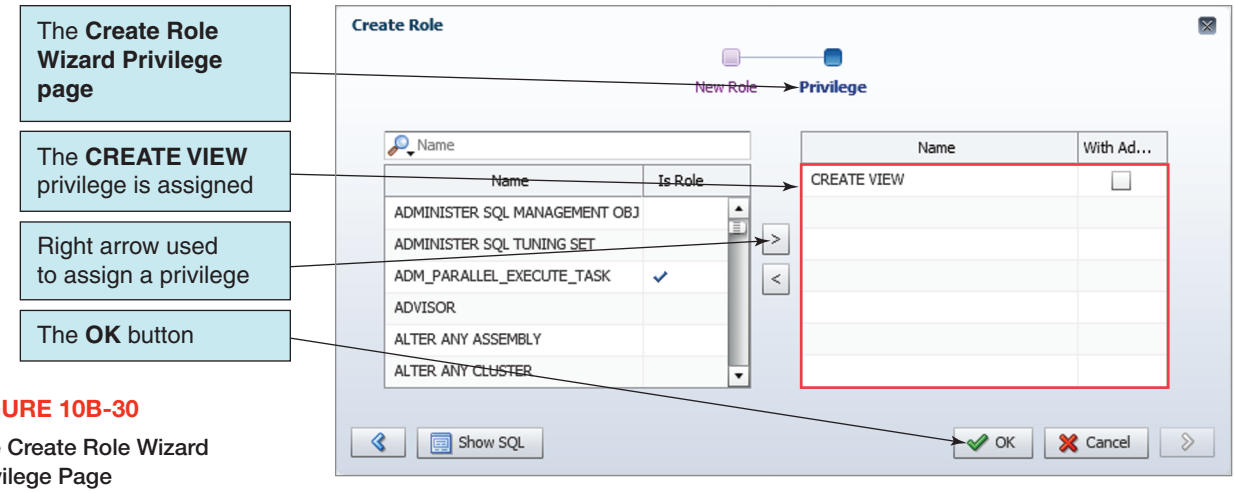


FIGURE 10B-30

The Create Role Wizard Privilege Page

Oracle Database SQL*Plus

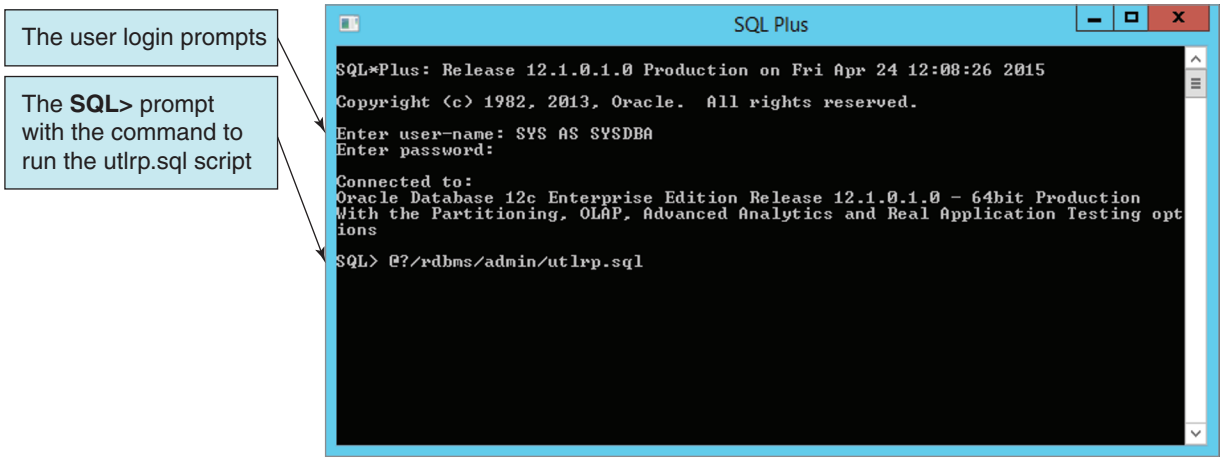
In the early development of computer technology, users interacted with computers using command-line utilities. A **command-line utility** is strictly text based. You are presented with a symbolic prompt to show you where to enter your commands. You type in a command (only one at a time) and press the [Enter] key to execute it. The results are displayed as plain-text (with some rudimentary character-based line and box-drawing capabilities) in response. All major computer operating systems have their version of a command-line utility. For personal computer users using a Microsoft operating system, the classic example is the MS-DOS command line, which still exists in Windows as the CMD program.

For Oracle Database, the classic command-line tool is **SQL*Plus**, which is still part of Oracle Database 12*c* and Oracle Database Express Edition 11*g* Release 2. SQL*Plus can be used to submit both SQL and Oracle Database Procedure Language/SQL (PL/SQL) statements (discussed later in this chapter) as well as perform various maintenance and administrative functions. It has some statements and commands that are unique to it—it is not simply a shell for submitting SQL and PL/SQL commands to the database. Figure 10B-31 shows SQL*Plus ready to run the *utlrp.sql* script, which is a utility script Oracle recommends running after Oracle Database 12*c* has been installed on a computer using a Windows operating system.[9]

As shown in Figure 10B-31, we are then connected to the Oracle Database DBMS with the SYS account running as SYSDBA (with the password set for SYS during installation), and then we typed in the command for the script. When we press the Enter key, the script will

FIGURE 10B-31

Oracle Database SQL*Plus



[9]See Section 7.2, "Validating Invalid PL/SQL Modules," in the Oracle Database Installation Guide 12*c* (12.1) for Microsoft Windows at *http://docs.oracle.com/database/121/nav/portal_11.htm*.

be run, and if you have installed Oracle Database 12*c* yourself, you should run this script. Note that this script command should be used *only* immediately after installation of Oracle Database 12*c* on a Windows system, as specified in the documentation. To close SQL*Plus, we use the **exit** command.

Oracle SQL Developer

As beloved as the SQL*Plus utility is among Oracle Database users, Oracle has developed a very powerful GUI utility that is now installed as part of the Oracle Database 12*c* installation process. The utility, **Oracle SQL Developer**, is particularly useful for database development with Oracle Database 12*c* and Oracle Database XE. SQL Developer is similar to the Microsoft SQL Server Management Studio (discussed in Chapter 10A) and the MySQL 5.6 Workbench (discussed in Chapter 10C).

- For *Oracle Database 12c*, a version is installed as part of the DBMS installation, and we start it from an icon in the Apps window (we recommend that you pin this icon to the task bar). However, we recommend that you download and install the latest version of SQL Developer and use it instead.
- For *Oracle Database XE*, you will have to download and install SQL Developer separately. In this case, we start it by finding the ..\sqldeveloper\sqldeveloper.exe file in the installation folder (directory), right-clicking to pin this icon to the task bar, and then clicking the icon on the task bar. Now we can just start it from the task bar.

In order to use SQL Developer, you must first install the Java Developer Kit (JDK). The easiest and recommended way to do this is to install NetBeans with JDK 8 available at the Oracle Java SE Downloads page at *www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html#javasejdk* as discussed earlier in this chapter. NetBeans is the integrated developer environment (IDE) that we use for Web page development in Chapter 11, and full download and installation instructions are in Appendix I. Install the JDK and the NetBeans IDE, and then install SQL Developer.

When you start SQL Developer for the first time, a dialog box will prompt you for the location of the Java JDK java.exe file.[10] SQL Developer will also ask you to configure file type associations, and you should select all the available types.

Updated versions of SQL Developer can be downloaded from *www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html*, and we recommend that you download and install the current version. In this chapter, we are running SQL Developer version 4.0.3.16.84.

Figure 10B-32(a) shows SQL Developer with the **New/Select Database Connection** dialog box open (opened by clicking the **New Connection** button) and with the settings for connecting to the CAPE_CODD tablespace in the *orcl* database.

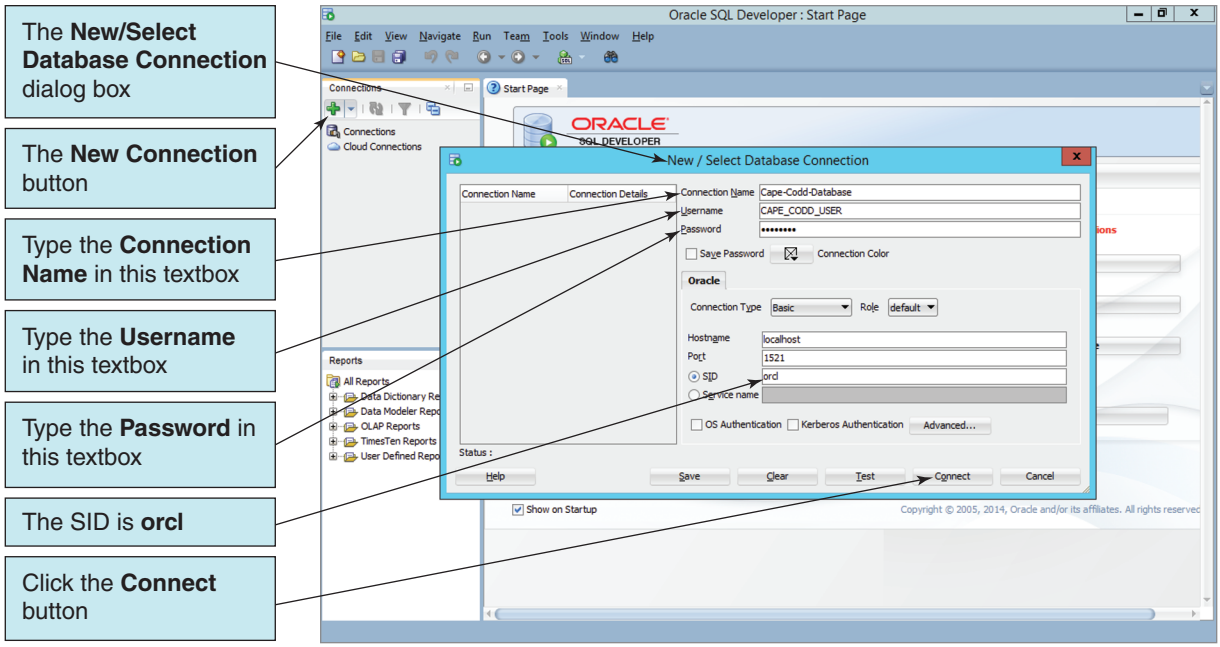
Figure 10B-32(b) shows SQL Developer with the **New/Select Database Connection** dialog box open (opened by clicking the **New Connection** button) and with the settings for connecting to the CAPE_CODD_USER workspace in the *xe* database.

Note that when Oracle asks for a **System Identifier (SID)**, it is asking for the database instance name, in this case:

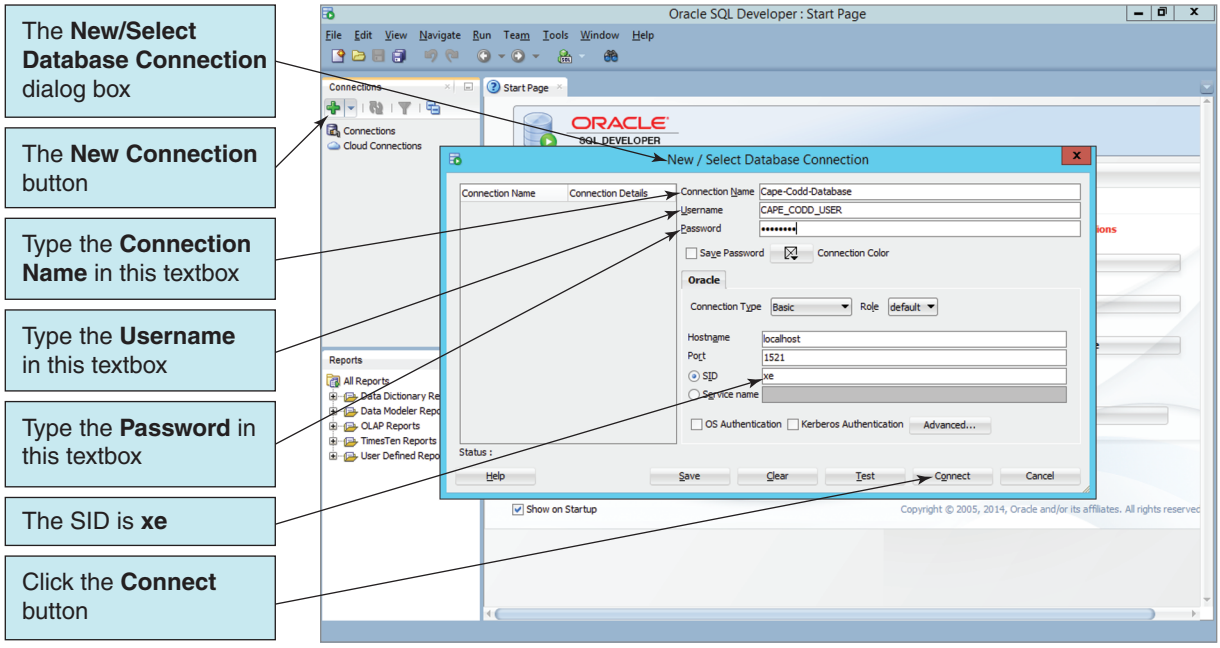
- For *Oracle Database 12c*, the SID is *orcl*.
- For *Oracle Database XE*, the SID is *xe*. Note: If this doesn't work in your installation of Oracle Database XE, try specifying a *service name* instead of an SID, and use the default service name of XEXDB.

From this point on in this chapter, we will be working in **Oracle Database XE using SQL Developer** unless otherwise noted. Note that everything we do can also be done in *Oracle Database 12c using SQL Developer*. While there would be some differences in the database objects displayed (in Oracle Database XE we will see all objects in the USER tablespace, while in Oracle Database 12c we would see the objects in the CAPE_CODD tablespace), the SQL statements work exactly the same, and the results will be the same.

[10] On a Windows operating system, look in C:\Program Files\Java\jdk{version number}\bin.



(a) Oracle Database 12*c* Cape-Codd-Database Connection



(b) Oracle Database XE Cape-Codd-Database Connection

FIGURE 10B-32
Oracle SQL Developer

In Figure 10B-33, CAPE_CODD_USER has connected to the USERS tablespace, and we can see the Cape-Codd-Database connection open in SQL Developer. Although we will use the Enterprise Manager Database Control for database administration in Oracle Database 12c and Oracle Database XE 11.2 for database administration in Oracle Database XE, we will use SQL Developer for database development.

Creating a Workspace for the SQL Developer Files

Before using the SQL Developer, we recommend creating a folder named *SQL Developer* under the Documents folder (or whatever your main data storage area is named). In Windows, this can be done using Windows Explorer, as shown in Figure 10B-34. In this workspace,

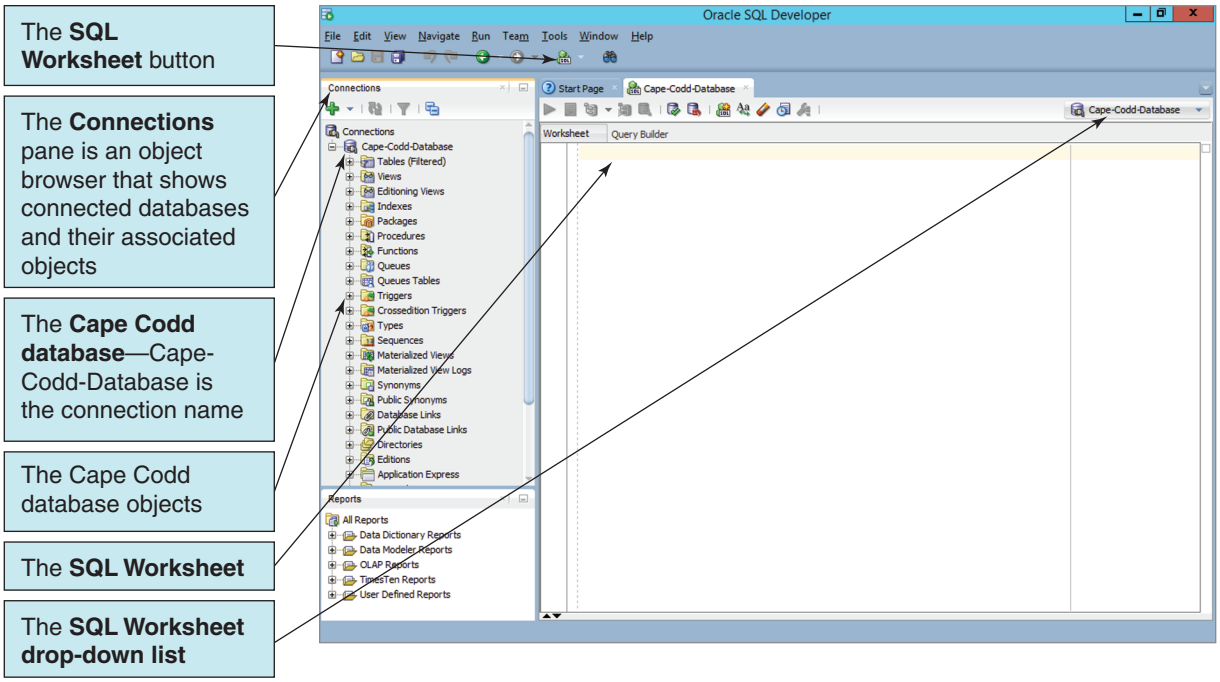


FIGURE 10B-33

The Cape Codd Database in SQL Developer

create a subfolder for each database project. At this point, we need to create a subfolder for the Cape Codd database we use in Chapter 2.

Oracle Database Schemas

We just described CAPE_CODD_USER as connecting to a tablespace (which holds the objects of the database application) in SQL Developer. However, although this is basically true, it is not quite what is actually happening. When CAPE_CODD_USER connects to the *orcl* database instance in Oracle Database 12c or the xe database instance in Oracle Database XE with SQL Developer, he sees the objects in the *CAPE_CODD_USER schema*. In Oracle Database, a **schema** is the collection of objects in the database instance that can be used by a user. When a new user is created, a new schema with the exact same name is created and associated with the user. When the user is given privileges to work with an object, that object becomes part of his schema. A schema may span tablespaces—if a user has rights to an object, it is in his or her schema. Therefore, when CAPE_CODD_USER connects to the database instance using SQL Developer, perhaps a more accurate connection name would be *CAPE_CODD_USER-Schema*.

Creating and Using an Oracle Database Database

Now that the Oracle Database DBMS is installed and we have SQL Developer open, we can create and develop a new database. We will create a database named *Cape_Codd* for the Cape Codd Outdoor Sports database we used in Chapter 2 to discuss SQL query statements.

Creating a Database in Oracle Database

In reality, we have already created the equivalent of a database in Oracle Database. The creation of tablespaces, users, roles, and schemas as discussed above is the Oracle Database equivalent of creating a Microsoft SQL Server database or a MySQL schema. In Oracle Database, what we call a *database* is a set of related objects stored in one or more tablespaces and spanned by one or more user schemas.

For convenience, we can label our SQL Developer connection with a name that includes the work database so that we know what database object collection we are using and working on in Oracle Database. Thus, we labeled the Cape Codd database connection as Cape-Codd-Database, and when we connect to Oracle Database using the connection, our intention is to work on Cape Codd database development.

BY THE WAY

Books on systems analysis and design often identify three design stages:

- Conceptual design (conceptual schema)
- Logical design (logical schema)
- Physical design (physical schema)

The creation and use of file structure and file organization (including physical storage placement and file characteristics) to store database components and physical records of data are a part of the *physical design*, which is defined in these books as the aspects of the database that are actually implemented in the DBMS. Besides physical record and file structure and organization, this includes indexes and query optimization.

Oracle Database SQL Statements and SQL Scripts

Because we have already argued that you need to know how to write and use SQL statements instead of relying on GUI tools, we come back to simply using SQL as the basis of our work. But we do not want to use a command-line utility, and we are not going to use the GUI tool in GUI mode, so what is left?

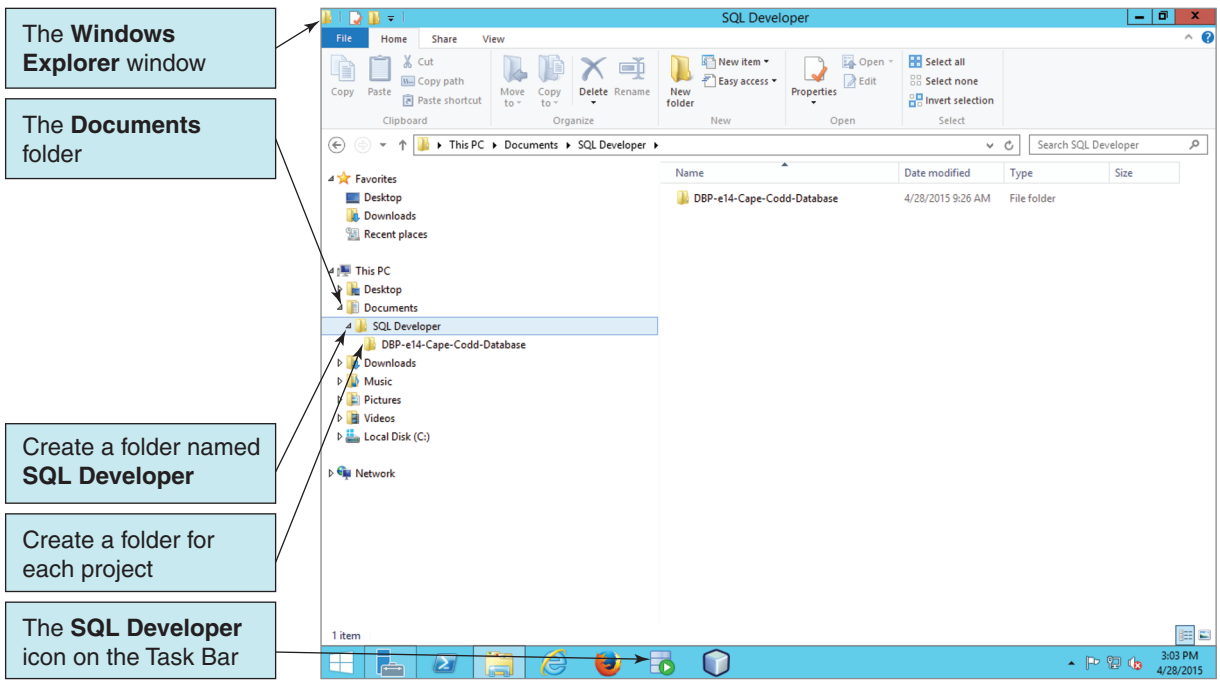
The answer is that the SQL Developer provides us with an excellent SQL editing environment. This lets us take advantage of GUI capabilities while still working with text-based SQL statements. We do this by opening and using an SQL Editor window in which to create and edit our SQL statements. Note that, when we connect to the Oracle Database server, the SQL Developer window is opened with a tabbed SQL Worksheet window by default. Thus, we already have one available. We can open others, as needed, by clicking the **SQL Worksheet** button (see Figure 10B-38).

The SQL editing environment in an SQL Worksheet window will be our tool of choice for editing SQL DDL and DML statements. One advantage of using this SQL editor is the ability to save and reuse SQL scripts. For Oracle Database, **SQL scripts** are plaintext files labeled with the **.sql* file extension. We can save, open, and run (and rerun) SQL scripts.

By default, Oracle Database will save scripts in the user's C:\Users\{UserName} folder. Because this does not separate Oracle Database files from other data files, we recommend using the folder structure we created earlier in this chapter and shown in Figure 10B-34 with an *SQL Developer* folder, with a separate folder for each database project, such as *DBP-e14-Cape-Codd-Database*, under it.

FIGURE 10B-34

The SQL Developer Folder in Windows Explorer



An SQL script is composed of one or more SQL statements, which can include SQL script comments. **SQL script comments** are lines of text that do not run when the script is executed but are used to document the purpose and contents of the script. Each comment line begins with the characters **/* (slash asterisk)** and ends with the characters ***/ (asterisk slash)**. For single-line comments, we proceed the text with **-- (two dashes)**.

Using Existing SQL Scripts

Another advantage of SQL scripts is that we can use scripts written by others. To do this, we simply store the script in an appropriate location on our computer, open it in an SQL script tabbed window, and execute it.

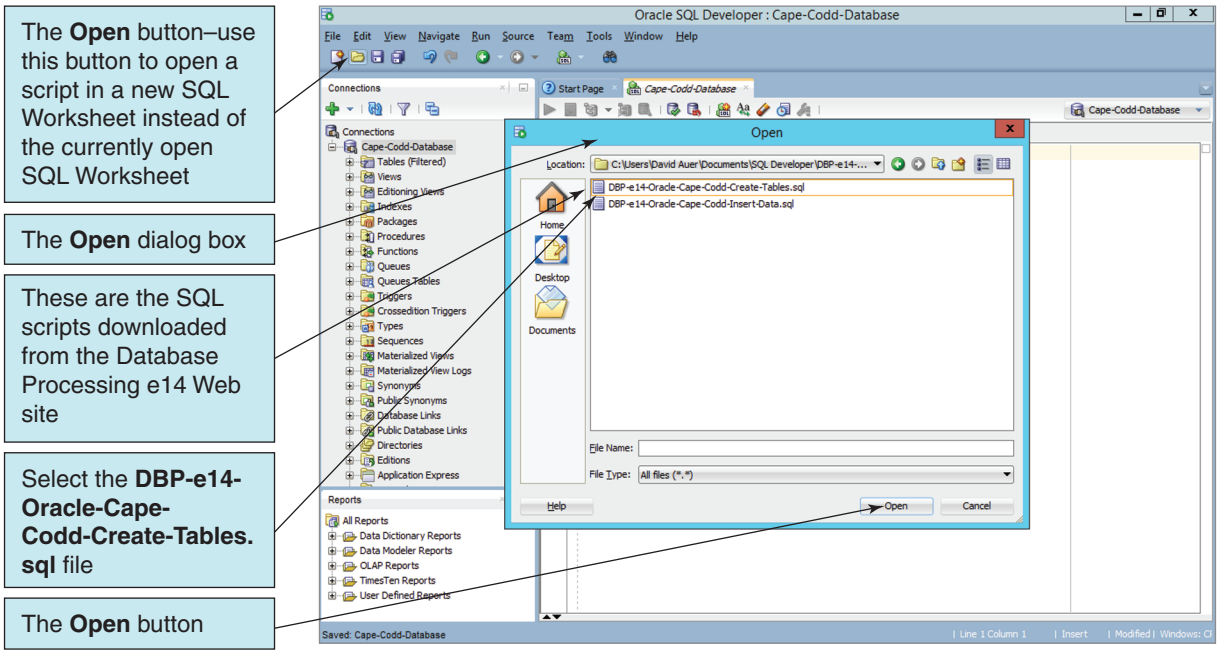
To illustrate this, we will build the Cape Codd database used in Chapter 2 to use with our example SQL query statements. This will then allow you to run the Chapter 2 statements in an actual database and to work the Review Questions at the end of that chapter.

Opening and Running an Existing SQL Script

1. The SQL scripts needed to build the Cape Codd database are available at *www.pearsonhighered.com/kroenke*. Go to the *Database Processing* 14/e Companion Web site, and download the Student Data Files to your *Downloads* folder. There is a ZIP archive file named **DBP-e14-IM-SRC.zip**, so you will need to extract the files. After you have done this, copy the two files in the *Downloads/SQL Developer/DBP-e14-Cape-Codd-Database* folder to your *Documents/SQL Developer/DBP-e14-Cape-Codd-Database* folder (if you have not already created the *DBP-e14-Cape-Codd-Database* folder in *SQL-Developer*, then copy the entire *Downloads/SQL Developer/DBP-e14-Cape-Codd-Database* folder to your *Documents/SQL Developer* folder).
2. Click the **Open** button shown in Figure 10B-35 to display the Open SQL Script dialog box (alternately, we can use the *File | Open* menu command to open the dialog box). The advantage of this button is that the script will be opened in a new query tabbed window, not the *Query 1* tabbed window that we currently have open.
3. Browse to the **DBP-e14-Oracle-Cape-Codd-Create-Tables.sql** SQL script as shown in Figure 10B-35.
4. Click the **Open** button. The DBP-e14-Oracle Database-Cape-Codd-Create-Tables SQL script is displayed in a new SQL worksheet tabbed window, as shown in Figure 10B-36.
5. Select the **Cape-Codd-Connection** in the Connection drop-down list, and then click the **Run Script** button. The SQL script is run, and the Cape Codd database

FIGURE 10B-35

The Open Dialog Box



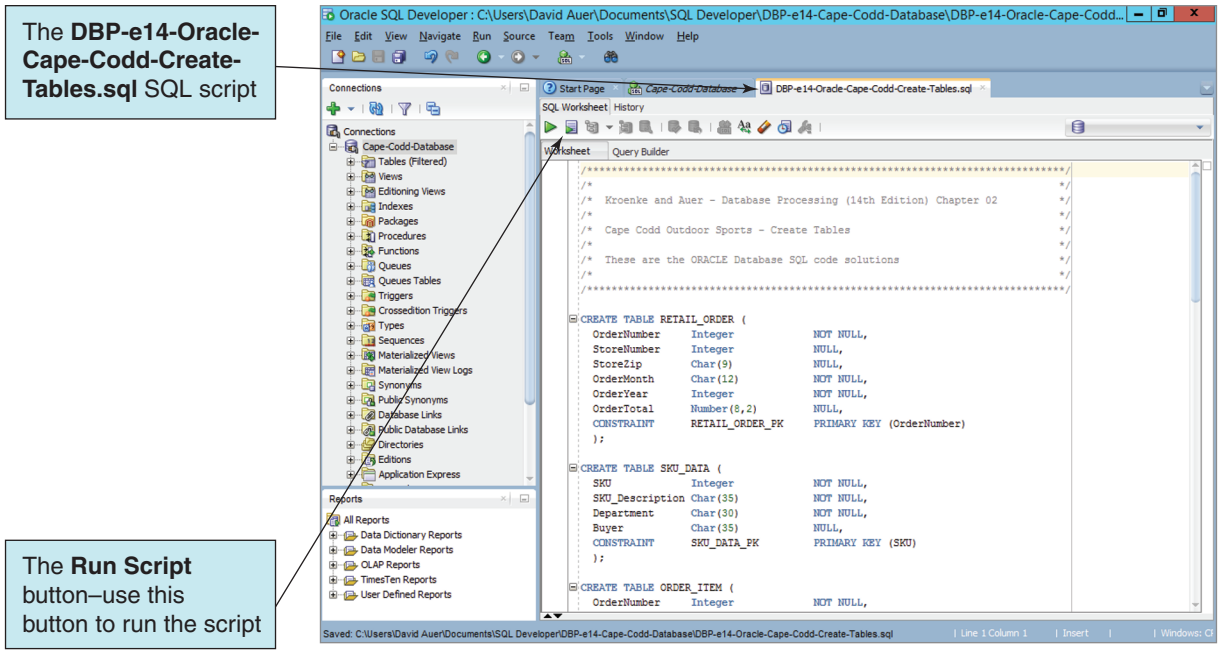


FIGURE 10B-36

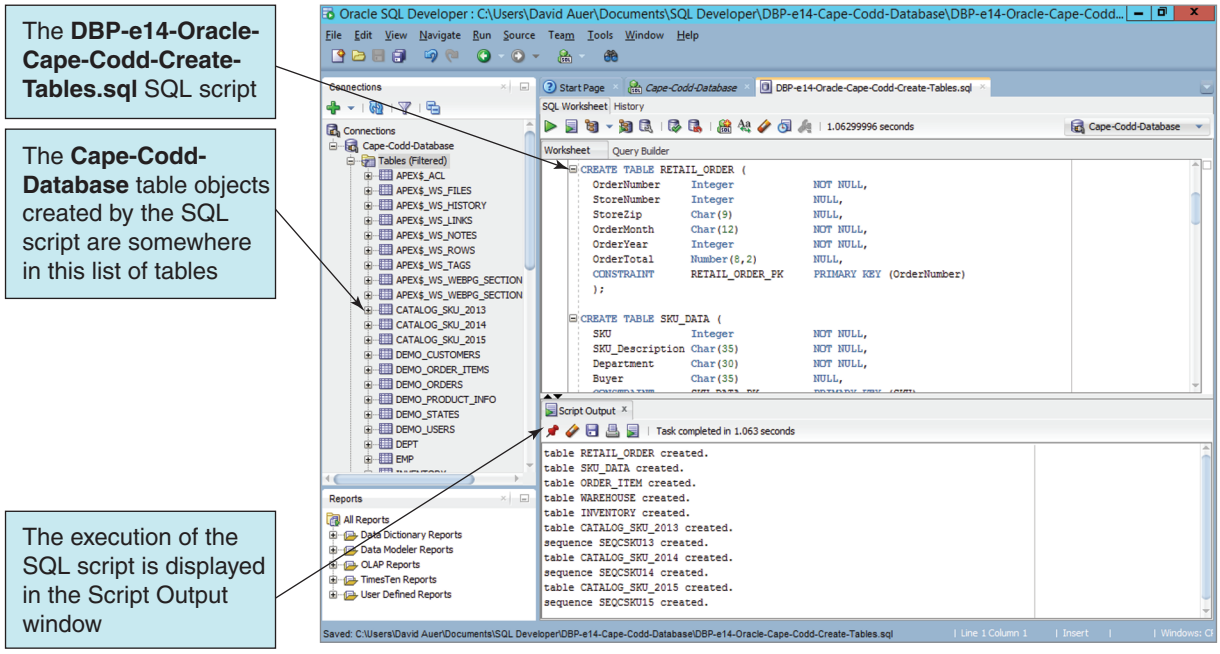
The Cape Codd Database Create Tables Script

tables and sequences are created, as shown in Figure 10B-37. Note that we have opened the Output window, which displays the script actions.

6. Note that there are too many Table objects displayed in in Figure 10B-37. This is because we are seeing everything in the USERS tablespace. We need to use a filter to control what is displayed. Right-click the Tables object in the Connections pane, and then click the **Apply Filter** button to open the Filter dialog box. Create the Table filters shown in Figure 10B-38.
7. Click the **OK** button in the Filter dialog box. The filtered tables now appear as shown in Figure 10B-39. Note that filters can be applied to various objects so that we see only the objects in the tablespace that are actually part of our database.
8. Click the **Open** button to display the Open dialog box.
9. Browse to the **DBP-e14-Oracle-Cape-Codd-Insert-Data.sql** SQL script.
10. Click the **Open** button. The DBP-e14-Oracle-Cape-Codd-Insert-Data.sql SQL script is displayed in a new SQL worksheet window.

FIGURE 10B-37

The Cape Codd Database Table Objects in the Navigator



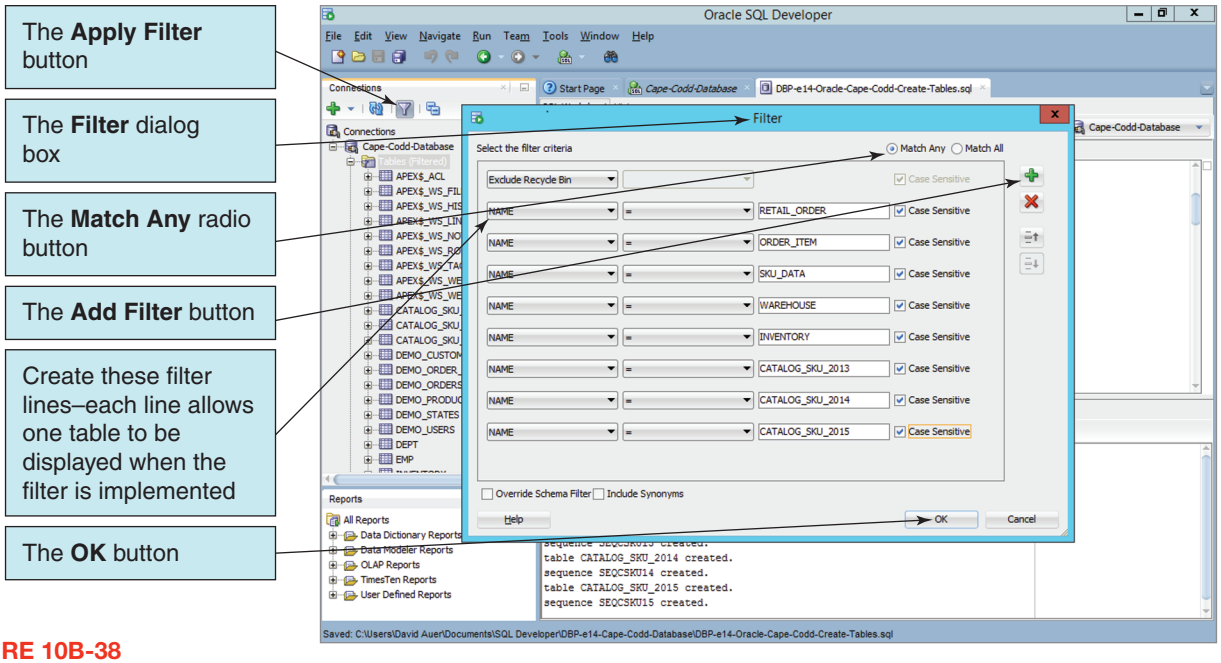


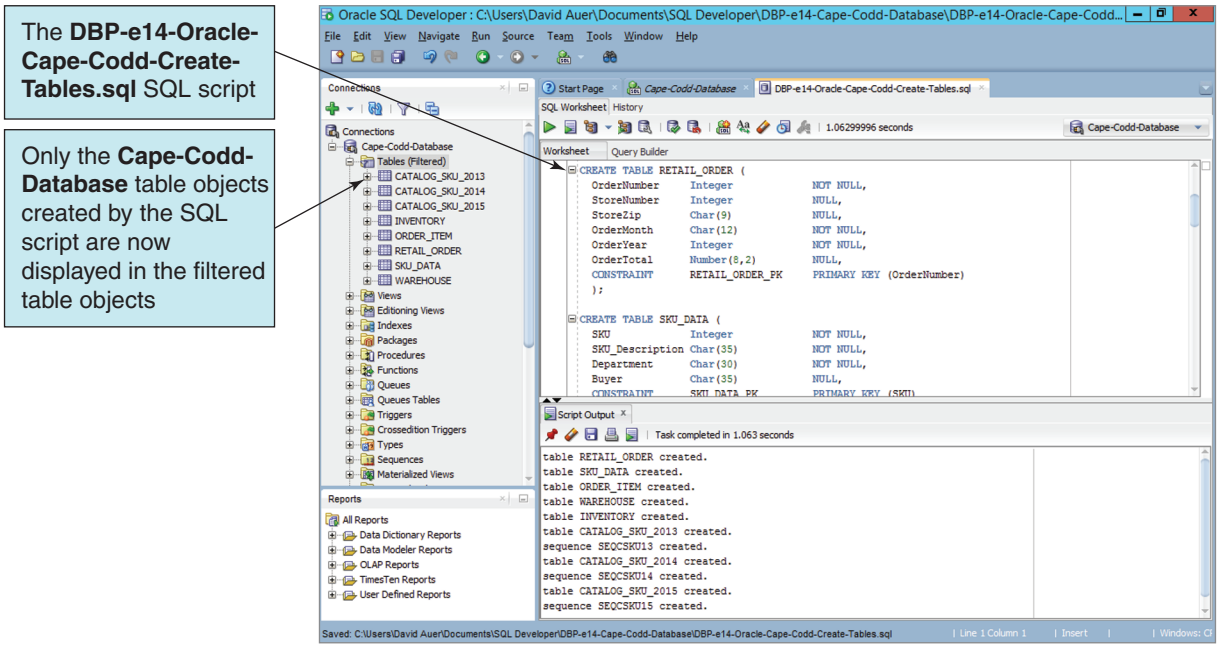
FIGURE 10B-38
The Filter Dialog Box

11. Select the **Cape-Codd-Connection** in the Connection drop-down list, and then click the **Run Script** button. The SQL script is run, and the Cape Codd database tables are populated.
12. In DBP-e14-Oracle-Cape-Codd-Insert-Data.sql SQL worksheet tabbed window, click the **X [Close]** button to close this tabbed window.
13. In DBP-e14-Oracle-Cape-Codd-Create-Tables.sql SQL worksheet tabbed window, click the **X [Close]** button to close this tabbed window.
14. We will now test the Cape Codd database by running a query against the database. As discussed in Chapter 2, we do this by using an SQL SELECT statement. We will run the first SQL query demonstrated in Chapter 2, which is:

```
SELECT    SKU, SKU_Description, Department, Buyer
FROM      SKU_DATA;
```

FIGURE 10B-39
The Filtered Cape Codd Database Tables

15. As shown in Figure 10B-40, enter the SQL statement for the SQL SELECT statement, and then click the **Run Statement** button. The SQL SELECT statement is run, and



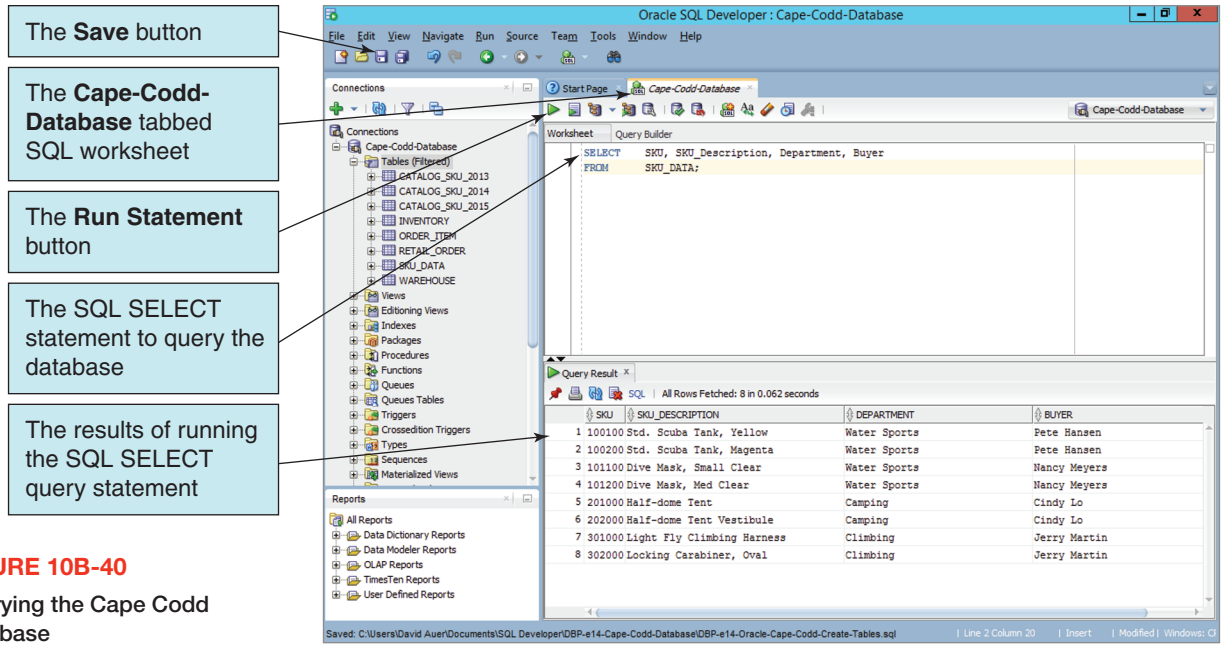


FIGURE 10B-40

Querying the Cape Codd Database

the query results are displayed in a tabbed Query Result window. These results confirm that the Cape Codd data was successfully entered into the Cape Codd database.

16. As discussed in Chapter 2, we can save this query as an SQL script for later use. Click the **Save** button to open the Save dialog box. Browse to the *Documents/SQL Developer/DBP-e14-Cape-Codd-Database* folder, as shown in Figure 10B-41, and save this SQL query as SQL-Query-CH02-01.sql.
17. In the Query Result tabbed window, click the **X [Close]** button to close this tabbed window.
18. In the SQL-QueryCH02-01.sql tabbed window, click the **X [Close]** button to close this tabbed window.

Using a Single SQL Script to Store Multiple SQL Commands

We have now created and queried the Cape Codd Outdoor Sports database used in Chapter 2 for our discussion of SQL query statements. We could save each of the Chapter 2 SQL queries

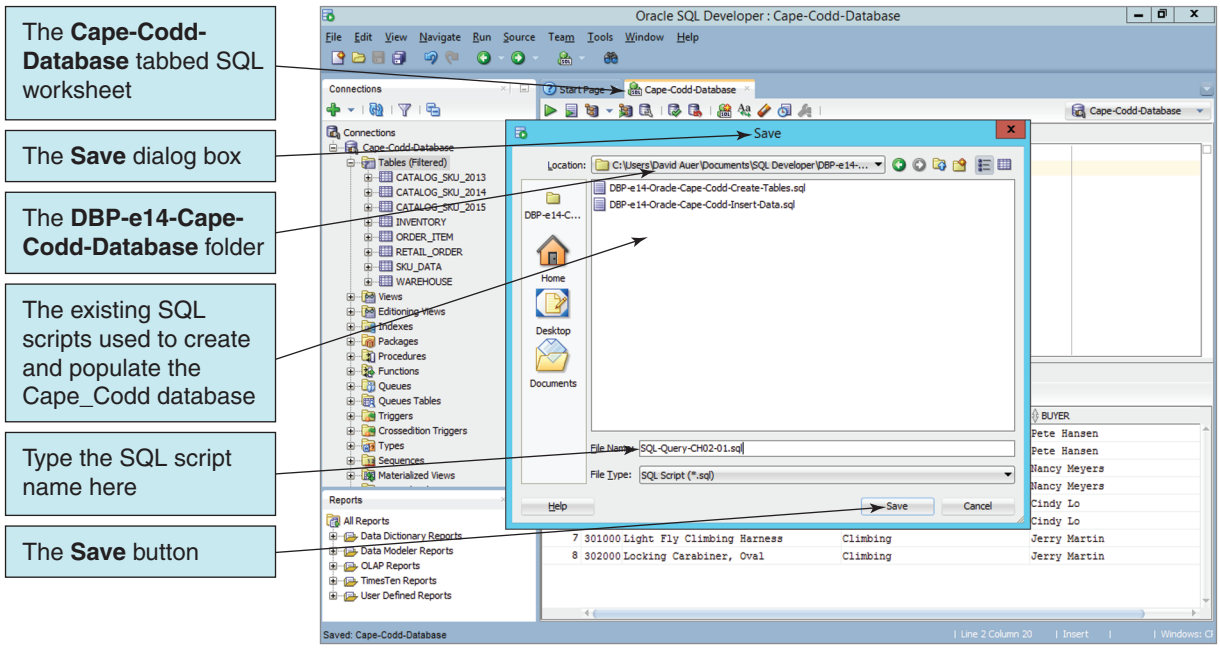


FIGURE 10B-41

Saving the SQL Query as an SQL Script

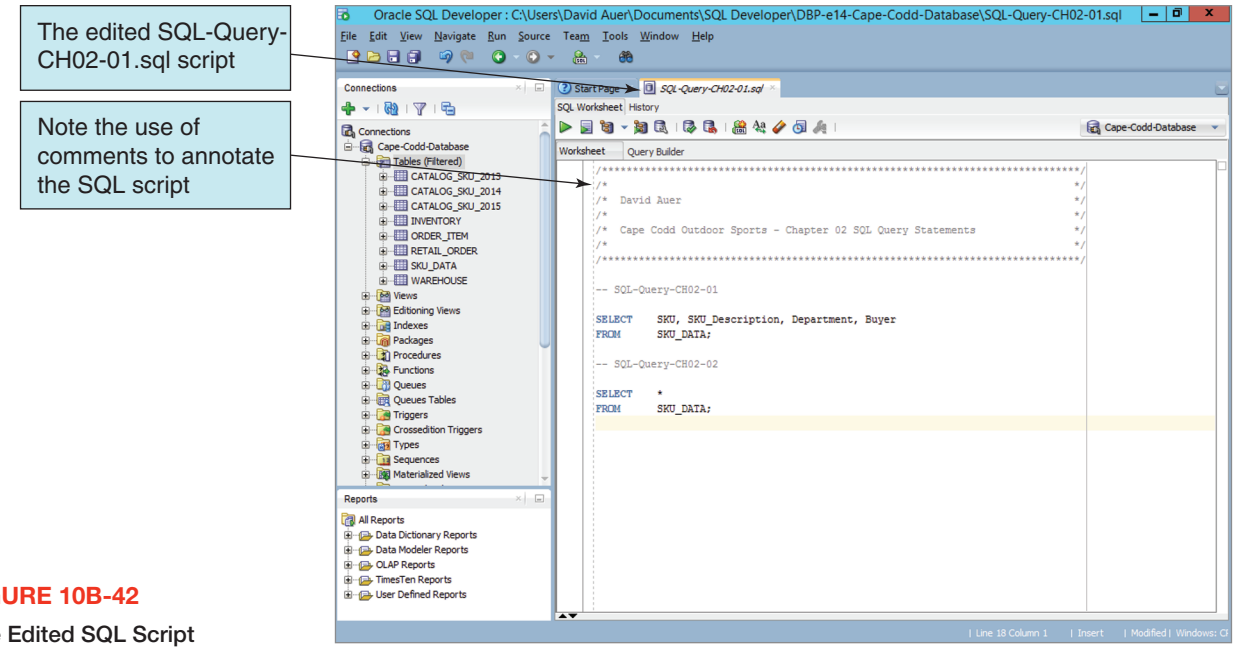


FIGURE 10B-42
The Edited SQL Script

as a separate SQL script, but a more efficient way to store these SQL statements is to combine them in a single SQL script.

We can annotate the SQL script with comments, and as we will demonstrate, we can run a single SQL statement in the script by selecting that statement and then executing it.

Creating and Using an SQL Script to Store SQL Queries

1. Click the **Open** button to display the Open SQL Script dialog box.
2. Browse to the **SQL-Query-CH02-01.sql** SQL script in the *Downloads/SQL Developer/DBP-e14-Cape-Codd-Database* folder.
3. Click the **Open** button. **SQL-Query-CH02-01.sql** SQL script is displayed in the SQL query tabbed window (although this is a new tabbed window, it will appear nearly identical to the tabbed window shown above in Figure 10B-40—the only differences will be that the tab is labeled *SQL-Query-CH02-01.sql* and the query has *not* been executed, so there is no Query Results window).
4. Edit the SQL script as shown in Figure 10B-42. Note that we are adding one new query (SQL-Query-CH02-02 from Chapter 2), an SQL comment header to identify the script (use your name in your script to identify it as yours!), and individual comments to identify each query.
5. While we could save our work under the same file name, the current name really doesn't describe the SQL script with the changes we have made. We will save it under a new, more descriptive name: *Cape-Codd-Chapter-02-SQL-Queries.sql*.
6. Use the **File | Save As…** command in the File menu as shown in Figure 10B-43 to display the **Save As** dialog box.
7. Type in the new SQL script name **Cape-Codd-Chapter-02-SQL-Queries.sql**.
8. Click the **Save** button to save the SQL script under the new file name. As shown in Figure 10B-44, instead of renaming the open tabbed SQL worksheet, SQL Developer opens the renamed script in a new tabbed SQL worksheet window displaying the new file name.
9. As shown in Figure 10B-44, use the cursor to select (highlight) **SQL-Query-CH02-02** (see how convenient comment labels are?).
10. Click the **Run Statement** button. Only the selected SQL command is executed, and the results are displayed as shown in the tabbed Results grid in Figure 10B-44. This illustrates how to select and run an individual SQL statement in an SQL script that contains many SQL statements.

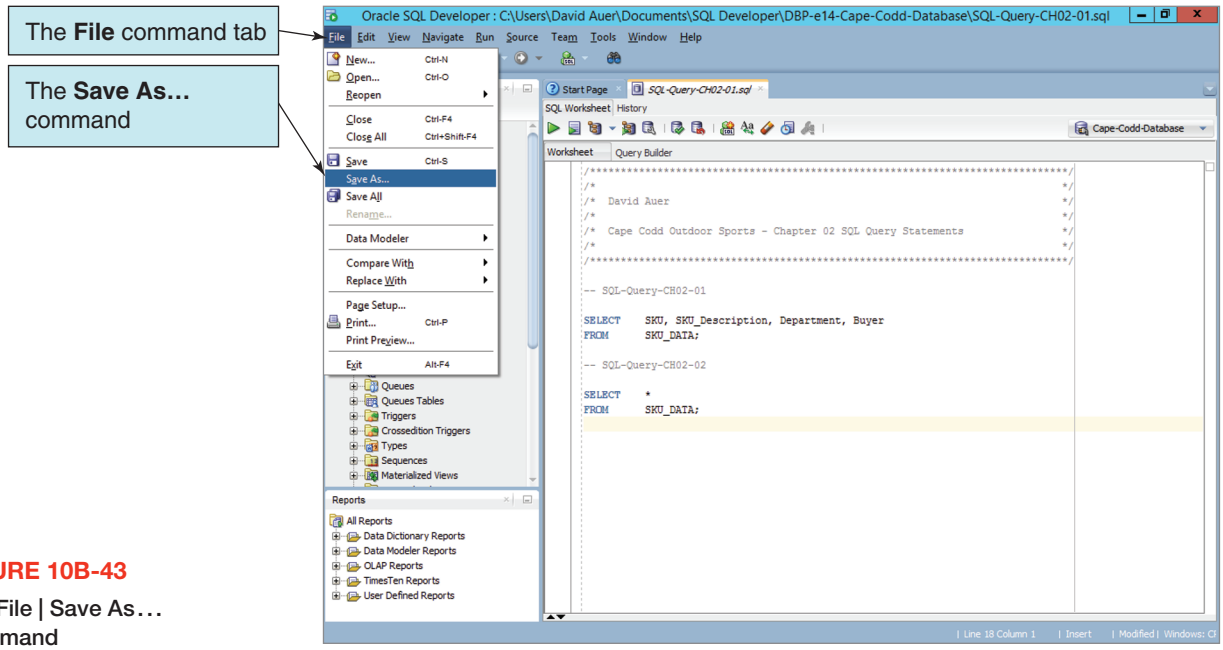


FIGURE 10B-43
The File | Save As… Command

11. Click the **X [Close]** buttons to close both the Cape-Codd-Chapter-02-SQL-Queries.sql and SQL-Query-CH02-01.sql tabbed windows. When asked if you want to save the changes to the SQL-Query-CH02-01.sql tabbed window, click the **No** button.

BY THE WAY

At this point, you have covered all the material about Oracle Database that you need to work with the SQL query statements in Chapter 2. If you worked through this material because of the directions in the **Using SQL in Oracle Database** section in that chapter, you should return to that section at this time and continue your work on SQL query statements. Use the new **Cape-Codd-Chapter-02-SQL-Queries.sql** script to store all your work in Chapter 2—it will be much easier and more efficient than storing a separate SQL script for each query!

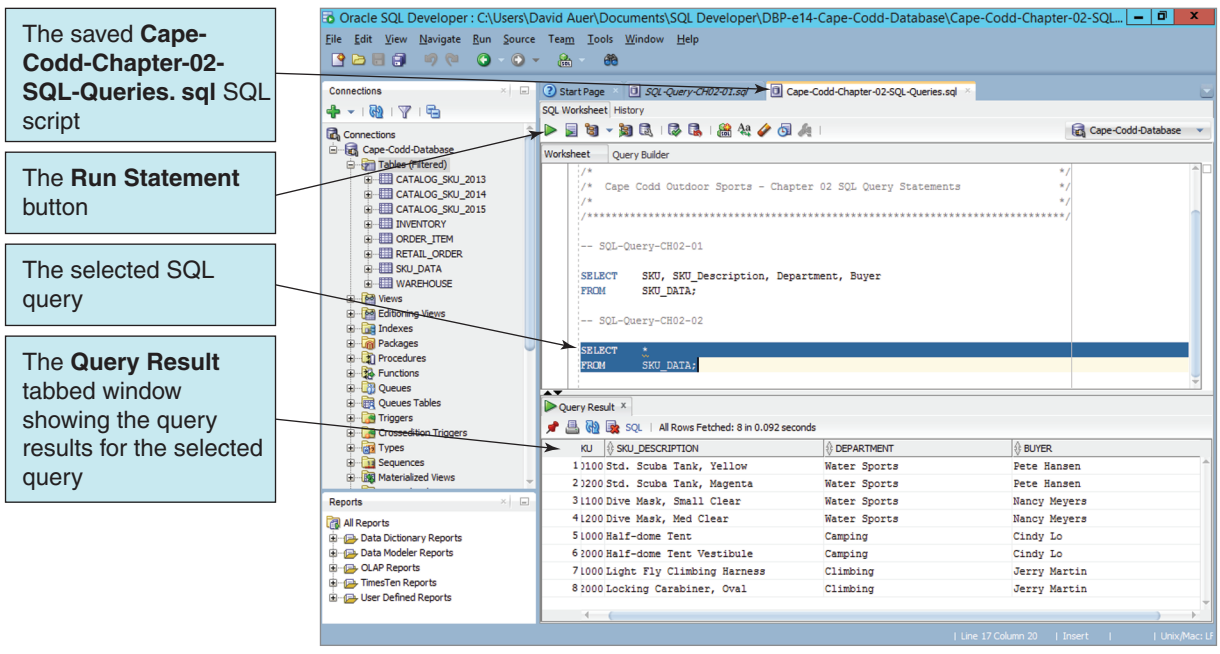


FIGURE 10B-44
The Renamed SQL Script

Implementing the View Ridge Gallery *VRG* Database in Oracle Database

Now that we know how to use existing SQL scripts and how to create and save SQL scripts for SQL query statements, we will discuss how to use SQL scripts to create and populate database tables of our own. To illustrate this, we will use the View Ridge Gallery *VRG* database introduced in Chapter 6 in our discussion of *database designs* and used as our example of database *implementation* in Chapter 7. In this chapter, we will discuss the specific implementation of the *VRG* database in Oracle Database and use that implementation to introduce some topics not covered in Chapter 7.

BY THE WAY

Because the VRG database example we use in Chapter 7 and this chapter is fairly complex, complete SQL scripts to create the VRG tables and populate them with data are available at the book's Web site at *www.pearsonhighered.com/kroenke*. These scripts will allow you to build the basic VRG database, and then actually try out the VRG database SQL code examples in the chapters. You will still need to read and understand the discussions of the SQL code for these two scripts to be sure you understand all the underlying concepts.

As we have seen, tables and other Oracle Database structures can be created and modified in two ways. The first is to write SQL code using either the CREATE or ALTER SQL statements we discussed in Chapter 7. The second is to use the Oracle Database GUI display tools discussed earlier in this chapter. Although either method will work, CREATE statements are preferred for the reasons described in Chapter 7. Some professionals choose to create structures via SQL but then modify them with the GUI tools.

Each DBMS product has its own variant of SQL, and each variant usually includes procedural extensions. The Oracle variant is called **Procedural Language/SQL** or **PL/SQL**. We will point out specific Oracle Database PL/SQL syntax as we encounter it in our discussion. For more on Oracle Database SQL and PL/SQL, see the Oracle Database Documentation SQL Language Reference and PL/SQL Language Reference at *http://docs.oracle.com/database/121/nav/portal_4.htm*.

First, we need to create the *VRG* database itself:

- For *Oracle Database 12c*, we use the same steps discussed above to create a VRG tablespace, a VRG_USER user account, and a VRG_DEV role.
- For *Oracle Database XE*, we will use the same steps discussed above to create a VRG_USER Oracle Application Express workspace.

Once we have created the basis for a new connection, we create a VRG-Database connection in SQL Developer. The New/Select Database Connection dialog box to create the new VRG-Database connection is shown in Figure 10B-45(a), and the established VRG-Database connection is shown in Figure 10B-45(b). As stated earlier in this chapter, we are using SQL Developer with Oracle Database XE throughout the rest of this chapter.

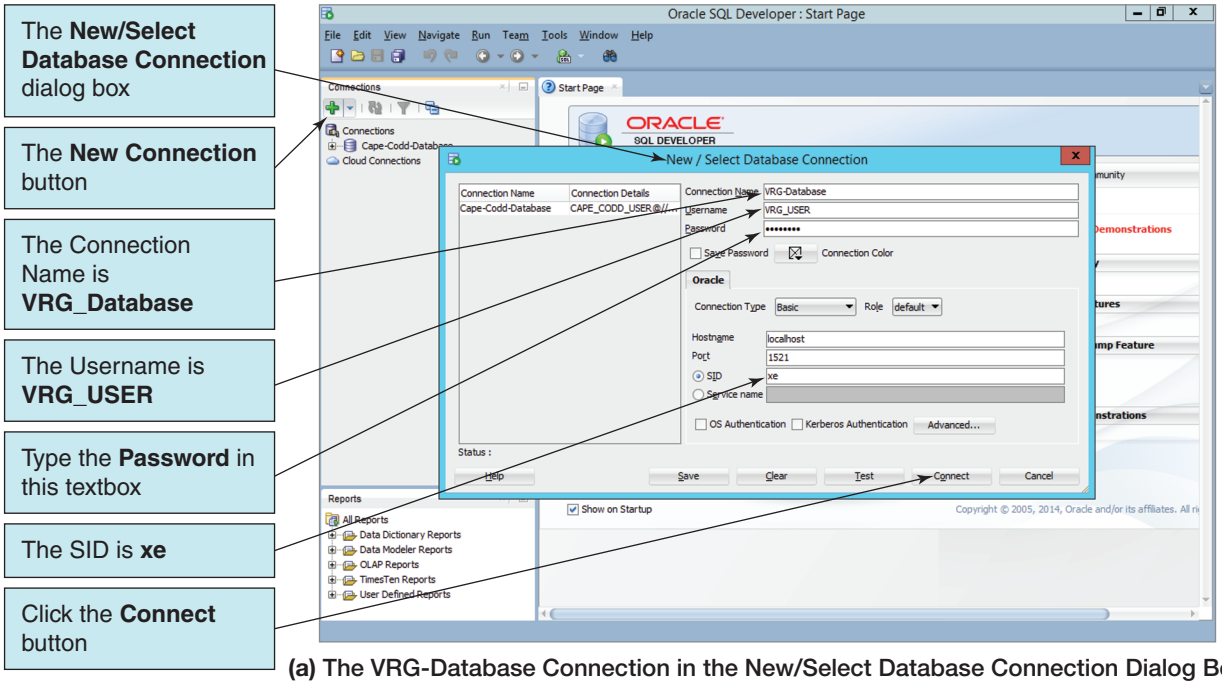
Using SQL Scripts to Create and Populate Database Tables

Now that we have created the VRG database, we will set up a *VRG-Database* folder in the *SQL Developer* folder to store our SQL scripts and review creating and saving an SQL script.

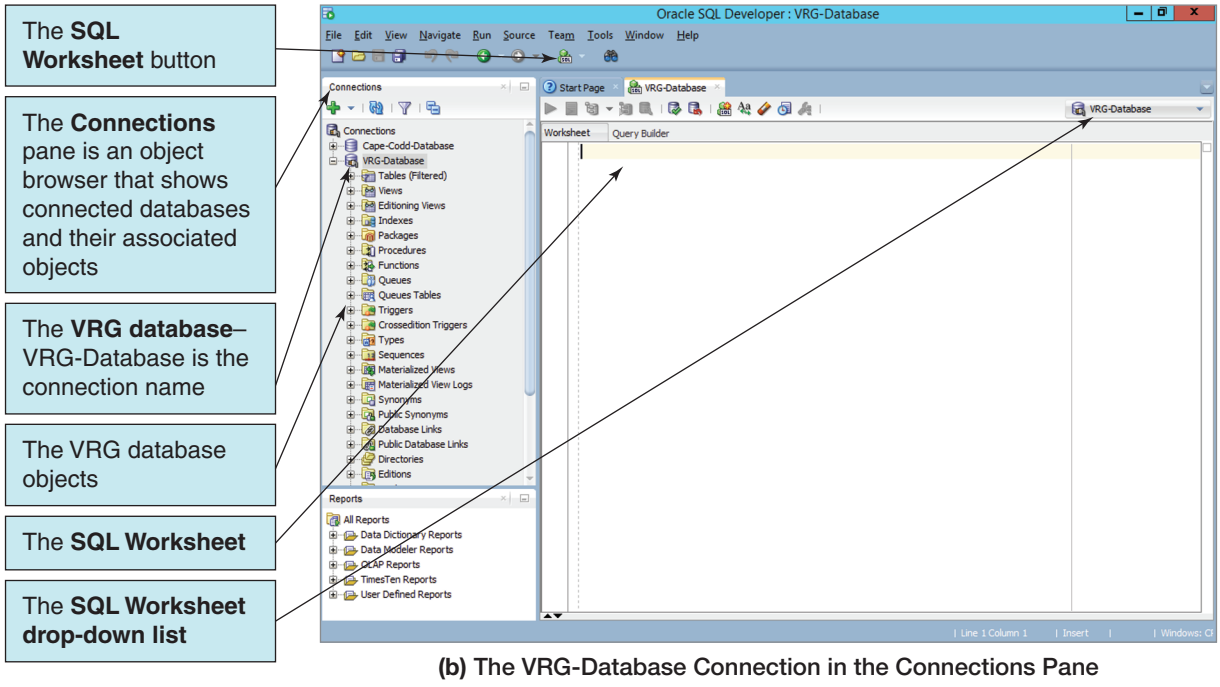
One advantage of using this SQL editor is that it enables us to save and reuse SQL scripts. For Oracle Database, *SQL scripts* are plaintext files labeled with the **.sql* file extension. We can save, open, and run (and rerun) SQL scripts. An SQL script is composed of one or more SQL statements, which can include SQL script comments. *SQL script comments* are lines of text that do not run when the script is executed but are used to document the purpose and contents of the script. Each comment line begins with the characters */* (slash asterisk)* and the ends with the characters **/ (asterisk slash)*. For single-line comments, we proceed the text with *-- (two dashes)*.

Creating and Saving an SQL Script to Create the VRG Tables

1. SQL Developer always opens a new SQL worksheet window when a connection is opened.



(a) The VRG-Database Connection in the New/Select Database Connection Dialog Box



(b) The VRG-Database Connection in the Connections Pane

FIGURE 10B-45

The VRG Database in SQL Developer

2. In the open tabbed SQL worksheet window, type the SQL comments shown in Figure 10B-46.
3. Click the **Save** button. The Save dialog box is displayed.
4. In the Save dialog box, browse to the *Documents/SQL Developer/DBP-e14-Cape-Codd-Database* folder. Browse up one level to the *Documents/SQL Developer* folder.
5. Click the **Create new subdirectory** button in the Save dialog box. The Create New Directory dialog box is displayed.
6. Type the folder name **DBP-e14-View-Ridge-Gallery-Database** as the new directory name.
7. Click the OK button in the Create New Directory dialog box. The new subdirectory is created, and the Save dialog box is directed to this new subdirectory.

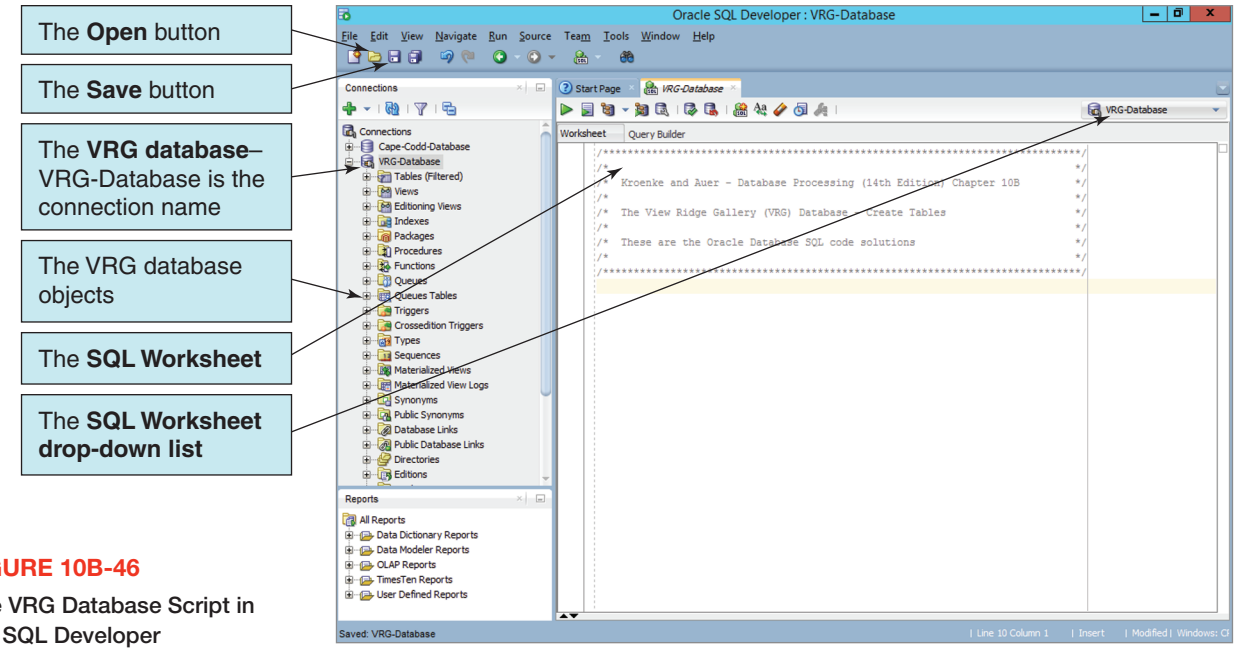


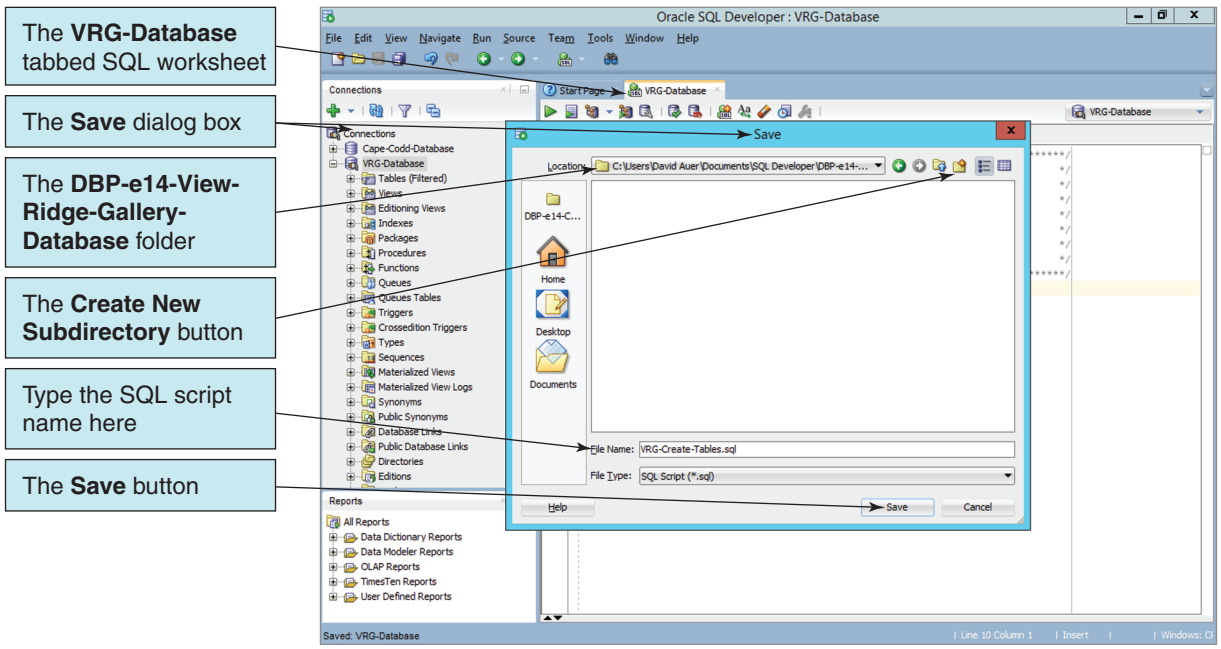
FIGURE 10B-46
The VRG Database Script in the SQL Developer

8. Type the file name **VRG-Create-Tables** in the File Name text box of the Save dialog box, as shown in Figure 10B-47.
9. Click the **Save** Button on the Save dialog box. The script is saved.

Creating the View Ridge Database Table Structure

The Oracle Database version of the SQL CREATE TABLE statements for the View Ridge Gallery VRG database in Chapter 7 are shown in Figure 10B-48. As discussed in Chapter 6, we are using the table name TRANS instead of TRANSACTION in Figure 10B-48. This was done because TRANSACTION is a **reserved word** in some of the databases discussed in this book.[11] Even if you make TRANSACTION into a **delimited identifier** by placing

FIGURE 10B-47
Saving the SQL Script



[11]For a complete list of Oracle Database and PL/SQL reserved keywords, see *http://download.oracle.com/docs/cd/E11882_01/appdev.112/e10830/appb.htm#CJHIIICD.*

```
/*****************************************************************************/
/*                                                                           */
/*     Kroenke and Auer - Database Processing (14th Edition) Chapter 10B     */
/*                                                                           */
/*     The View Ridge Gallery (VRG) Database - Tables                        */
/*                                                                           */
/*     These are the Oracle Database SQL code solutions                      */
/*                                                                           */
/*****************************************************************************/

CREATE TABLE ARTIST (
       ArtistID            Int              NOT NULL,
       LastName            Char(25)         NOT NULL,
       FirstName           Char(25)         NOT NULL,
       Nationality         Char(30)         NULL,
       DateOfBirth         Number(4)     NULL,
       DateDeceased        Number(4)     NULL,
       CONSTRAINT    ArtistPK            PRIMARY KEY(ArtistID),
       CONSTRAINT    ArtistAK1           UNIQUE(LastName, FirstName),
       CONSTRAINT    NationalityValues   CHECK
                           (Nationality IN ('Canadian', 'English', 'French',
                            'German', 'Mexican', 'Russian', 'Spanish',
                            'United States')),
       CONSTRAINT    BirthValuesCheck    CHECK (DateOfBirth < DateDeceased),
       CONSTRAINT    ValidBirthYear      CHECK
                           ((DateOfBirth >= 1000) AND (DateOfBirth <= 2100)),
       CONSTRAINT    ValidDeathYear      CHECK
                           ((DateDeceased >= 1000) AND (DateDeceased <= 2100))
       );

CREATE TABLE WORK (
       WorkID              Int              NOT NULL,
       Title               Char(35)         NOT NULL,
       Copy                Char(12)         NOT NULL,
       Medium              Char(35)         NULL,
       Description         Varchar(1000)    NULL,
       ArtistID            Int              NOT NULL,
       CONSTRAINT    WorkPK              PRIMARY KEY(WorkID),
       CONSTRAINT    WorkAK1             UNIQUE(Title, Copy),
       CONSTRAINT    ArtistFK            FOREIGN KEY(ArtistID)
                          REFERENCES ARTIST(ArtistID)
       );

CREATE TABLE CUSTOMER (
       CustomerID          Int              NOT NULL,
       LastName            Char(25)         NOT NULL,
       FirstName           Char(25)         NOT NULL,
       EmailAddress        Varchar(100)     NUll,
       EncryptedPassword   Varchar(50)      NUll,
       Street              Char(30)         NULL,
       City                Char(35)         NULL,
       State               Char(2)          NULL,
       ZIPorPostalCode     Char(9)          NULL,
       Country             Char(50)         NULL,
       AreaCode            Char(3)          NULL,
       PhoneNumber         Char(8)          NULL,
       CONSTRAINT    CustomerPK          PRIMARY KEY(CustomerID),
       CONSTRAINT    EmailAK1            UNIQUE(EmailAddress)
       );
```

FIGURE 10B-48

The SQL Statements to Create the VRG Table Structure

(continued)

```
CREATE TABLE TRANS (
       TransactionID        Int              NOT NULL,
       DateAcquired         Date             NOT NULL,
       AcquisitionPrice     Number(8,2)      NOT NULL,
       AskingPrice          Number(8,2)      NULL,
       DateSold             Date             NULL,
       SalesPrice           Number(8,2)      NULL,
       CustomerID           Int              NULL,
       WorkID               Int              NOT NULL,
       CONSTRAINT    TransPK             PRIMARY KEY(TransactionID),
       CONSTRAINT    TransWorkFK         FOREIGN KEY(WorkID)
                           REFERENCES WORK(WorkID),
       CONSTRAINT    TransCustomerFK     FOREIGN KEY(CustomerID)
                           REFERENCES CUSTOMER(CustomerID),
       CONSTRAINT    SalesPriceRange  CHECK
                          ((SalesPrice > 0) AND (SalesPrice <=500000)),
       CONSTRAINT    ValidTransDate      CHECK (DateAcquired <= DateSold)
       );

CREATE TABLE CUSTOMER_ARTIST_INT(
       ArtistID             Int              NOT NULL,
       CustomerID           Int              NOT NULL,
       CONSTRAINT    CAIntPK             PRIMARY KEY(ArtistID, CustomerID),
       CONSTRAINT    CAInt_ArtistFK      FOREIGN KEY(ArtistID)
                           REFERENCES ARTIST(ArtistID)
                              ON DELETE CASCADE,
       CONSTRAINT    CAInt_CustomerFK    FOREIGN KEY(CustomerID)
                           REFERENCES CUSTOMER(CustomerID)
                              ON DELETE CASCADE
       );
```

FIGURE 10B-48
Continued

the name in square brackets, as in [TRANSACTION], or some other delimiter, the DBMS may still become confused when executing the logic of stored procedures and triggers. Life became much simpler for applications using this database when the table TRANSACTION was renamed to TRANS. WORK is not currently a PL/SQL reserved word, but it is an ODBC reserved word (ODBC will be discussed in Chapter 11). Still, Oracle Database is less sensitive to it, and therefore we can use it.

Several alterations to the SQL statements shown in Chapter 7 had to be made for Oracle Database. Data types were modified to the Oracle Database data types shown in Figure 6-6(b). In addition, although Oracle Database supports ON DELETE actions, it does not support the ON UPDATE CASCADE constraint, and therefore all ON UPDATE constraints have been removed. The constraints on Nationality, DateOfBirth, and DateDeceased also were modified. The default value for Nationality was dropped. Because Oracle does not interpret the constraints shown in Chapter 7 for DateOfBirth and DateDeceased (LIKE '([1-2], [0-9], [0-9], [0-9])') correctly, range constraints were substituted. Also note that Oracle Database does not support surrogate keys as such. However, it does allow us to create a **sequence** and use values from that sequence in a primary key. We will define sequences when we discuss populating the tables. Other than these changes, the SQL in Figure 10B-48 should be very familiar to you by now.

Creating the VRG Table Structure Using SQL Statements

1. Click the **Open** button to display the Open dialog box.
2. Click the **Documents** button to open the Documents folder.
3. Double-click the **Oracle Workspace** folder icon to open it.
4. Double-click the **View-Ridge-Gallery-Database** folder icon to open it.
5. Click the **DBP-e14-VRG-Create-Tables.sql** script to select it, and then click the **Open** button to open the script in an SQL Worksheet.
6. You may be asked to reconnect to the VRG database at this point. If so, a dialog box will appear asking for the login name (we have been using VRG_USER) and password.

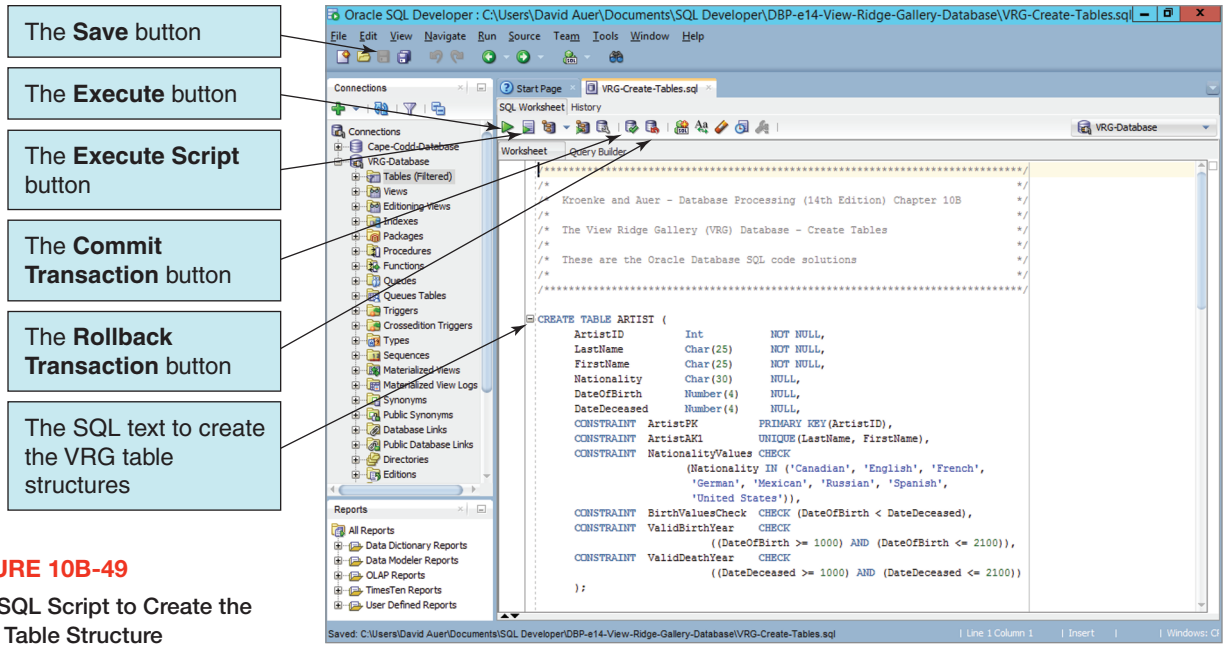


FIGURE 10B-49

The SQL Script to Create the VRG Table Structure

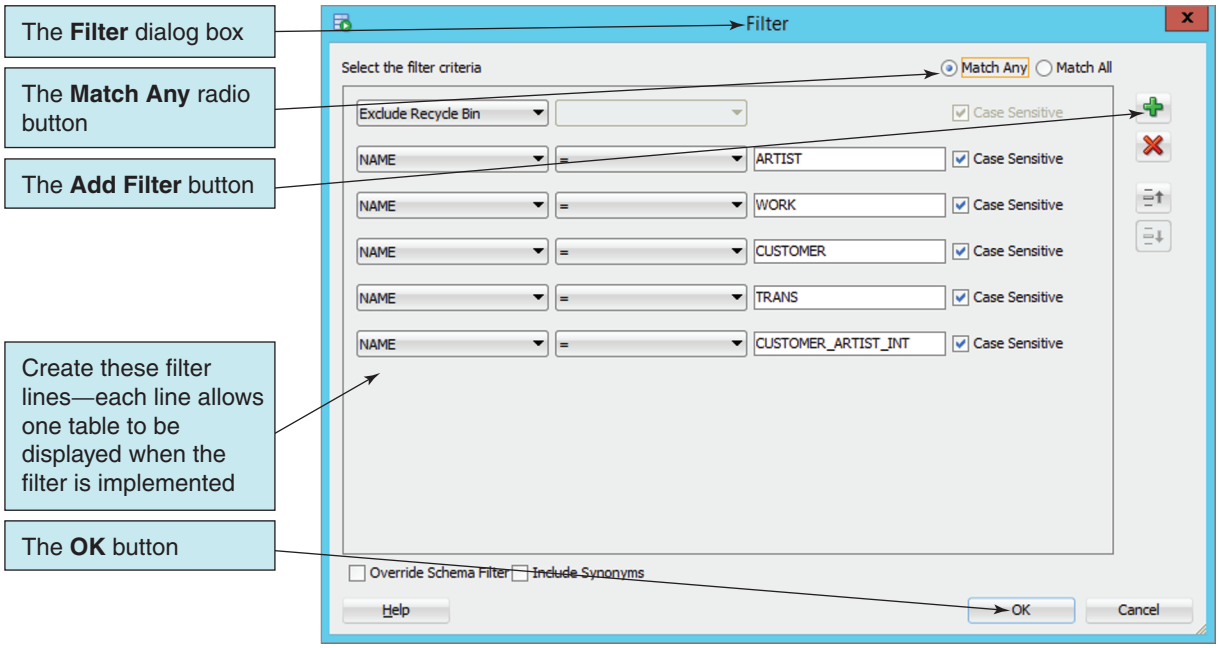
7. Type in the SQL statements shown in Figure 10B-48. Be sure to save the script often, and when you have completed entering all of the SQL statements, save the script a final time.
8. Scroll up to the top of the script. The completed SQL script to create the VRG table structure appears as shown in Figure 10B-49.
9. Be sure that the text insertion point is at the beginning of the script. Click the **Execute** button to run the script.
10. If there are any errors, messages appear in the Results pane below the SQL editor pane. If any error messages are displayed, then there are errors in your SQL statements. Correct any errors, and rerun the script. You can run individual SQL statements in the script by highlighting them and then clicking the **Execute** button.
11. Click the **Save** button to save your debugged SQL script.
12. Expand the **VRG-database Tables** folder to see the VRG tables.
13. Create and apply a table filter as shown in Figure 10B-50(a). Find the ARTIST table, and expand the ARTIST table to show the columns in the table, as shown in Figure 10B-50(b).
14. Close the SQL Worksheet containing the VRG-Create-Tables.sql script.

Now we have created the VRG table and relationship structure.

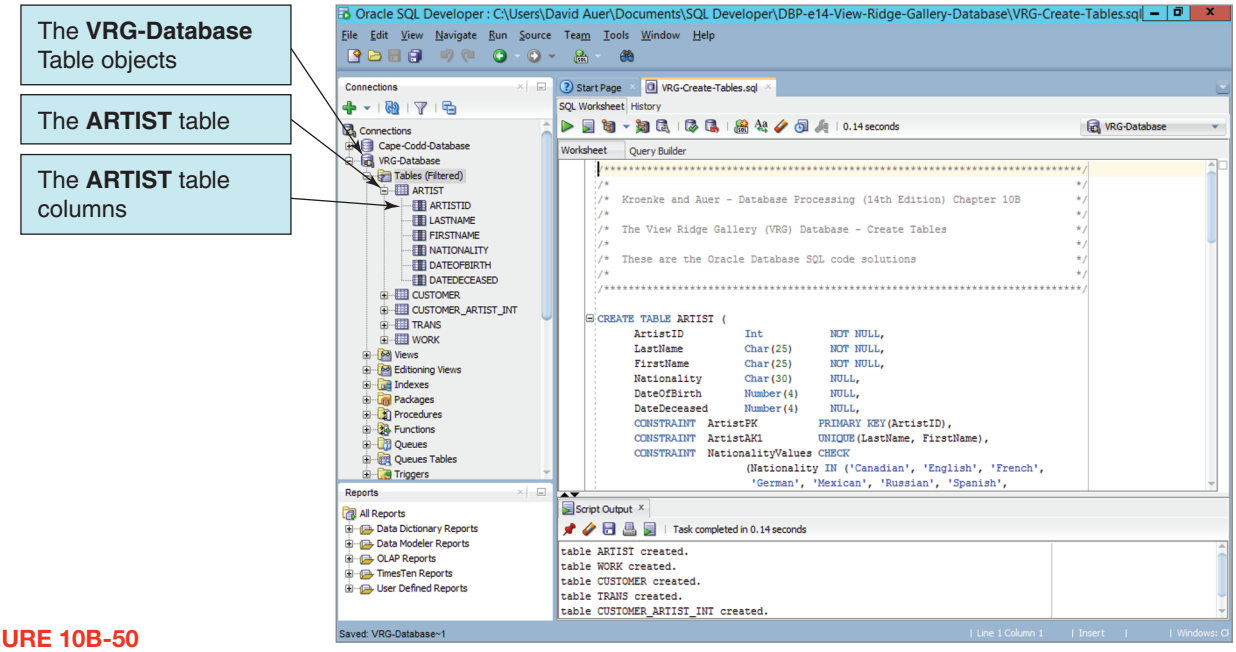
Transaction COMMIT in Oracle Database

Figure 10B-49 contains callouts to a **Commit Transaction button** and a **Rollback Transaction button**. In Chapter 9, we discussed transaction processing, where to ensure database integrity a transaction is initiated with (in Oracle Database) an **SQL SET TRANSACTION statement** and then either (1) committed to the database with an **SQL COMMIT statement** if the transaction is successful or (2) removed entirely from the database with an **SQL ROLLBACK statement** if there was an error in the transaction processing.

Transaction processing statements are used with DML commands such as INSERT and UPDATE. SQL Server 2014 and MySQL 5.6 implement implicit COMMITs, where a successful transaction is committed to the database automatically. Oracle Database, however, requires an explicit COMMIT command before the database changes are finalized.



(a) The VRG Database Table Filter



(b) The VRG Database Tables with ARTIST Expanded

FIGURE 10B-50
The VRG Database Tables

Therefore, when you run SQL DML statements that change the database data in Oracle Database, you must either (1) click the **Commit Transaction** button (or **Rollback Transaction** button) after you run the DML command or (2) include an SQL COMMIT statement in an SQL script. While this is not necessary with the SQL DDL statements we just used to create the database table structure because Oracle Database SQL DDL statements implicitly commit, it will be necessary when we start populating the tables.

Note that these COMMIT rules apply *only* to Oracle Database itself. Some Oracle Database utilities can be set to implement an implicit COMMIT, which is referred to as AUTOCOMMIT. For a more detailed discussion, see the Oracle Database documentation or just ask Tom at *http://asktom.oracle.com/pls/asktom/f?p=100:11:0::::P11_QUESTION_ID:314816776423*.

Reviewing Database Structures in the SQL Developer GUI Display

After building the table structure using SQL statements, we can inspect the results using the SQL Developer GUI tools. We will take a look at the ARTIST table, particularly the properties of the ArtistID primary key.

Viewing the ARTIST Table Structure in the GUI Display

1. In the SQL Developer Connections pane object browser, click the **ARTIST** table object. The ARTIST table design is displayed in a tabbed document window, as shown in Figure 10B-51, with the columns and column properties shown.

BY THE WAY If you look carefully at Figure 10B-51, you will see that ArtistID is shown as Number (38,0). Recall, however, that in our SQL statements in Figure 10B-49 we specified this number as integer (Int). By definition, integers have no decimal places (i.e., zero decimal places). Thus, Number (38,0) does designate an integer, and this is how Oracle Database stores integers.

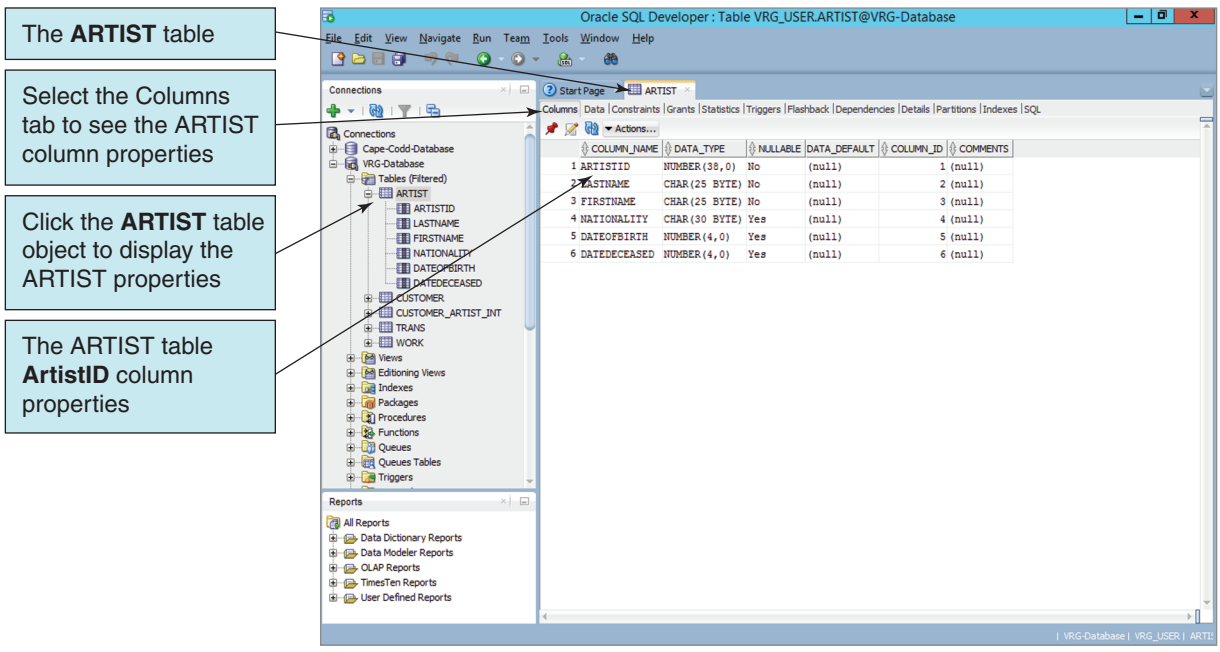
We can also inspect the constraints on the ARTIST table. Let's take a look at the ValidBirthYear constraint we coded into our SQL CREATE TABLE statements.

Viewing the ARTIST Table Constraints in the GUI Display

1. In the ARTIST tabbed document, click the **Constraints** tab to display the ARTIST table constraints. The ARTIST table constraint objects are displayed as shown in Figure 10B-52.
2. Note that the **ValidBirthYear** constraint is displayed and that the constraint is correct.
3. Close the **ARTIST** table tabbed pane in SQL Developer.

Note that in Figure 10B-52 there are also constraints for the NOT NULL status we set on ArtistID, LastName, and FirstName. Because we did not name them, Oracle has named them

FIGURE 10B-51
The ARTIST Table Columns and Column Properties



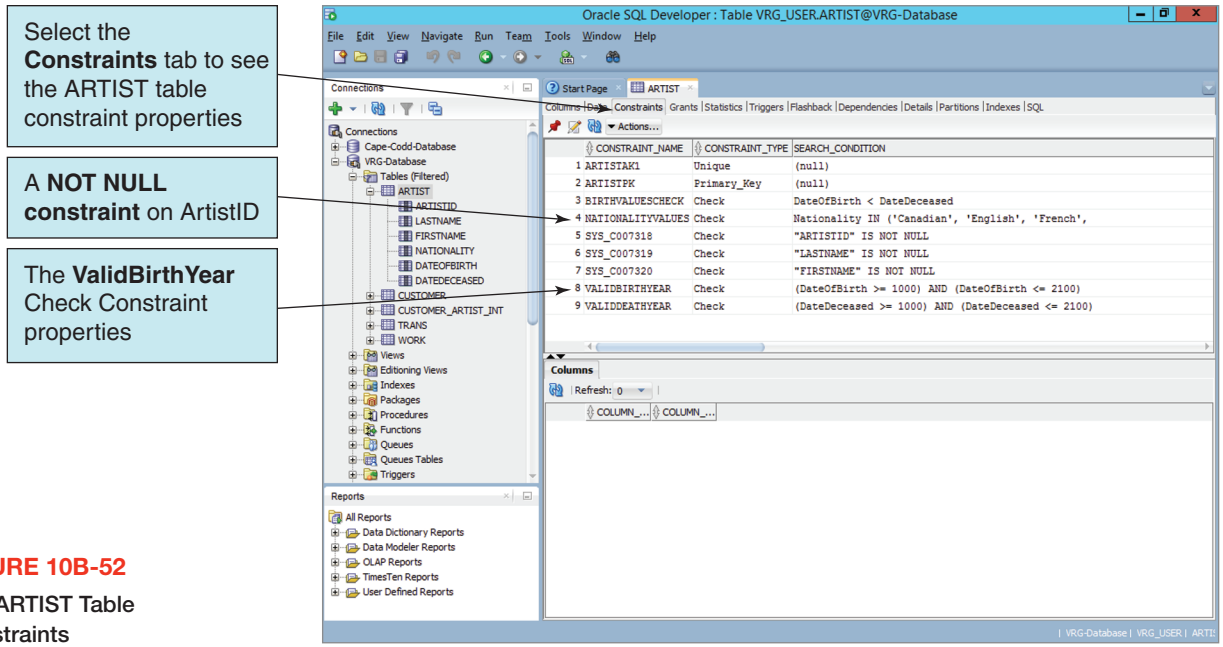


FIGURE 10B-52
The ARTIST Table Constraints

as SYS_C007318, SYS_C007319, and SYS_C007320, respectively. Given our preference for providing our own names for constraints, it looks like we should have provided names for the NOT NULL columns. In fact, in Oracle we could have done this. For example, the SQL syntax we would have used for ArtistID would have been:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
ArtistID    Int  CONSTRAINT ArtistID_Not_Null   NOT NULL
```

Another way to look at table properties is to use the Edit command in the table shortcut menu to open the Edit Table table editor (this tool is too complex to simply call it a dialog box). Edit Table displays more information and options than we have seen in the previous display (which is also opened by the Open command in the shortcut menu). Because ARTIST does not have any foreign keys, we will close it and look at the foreign key in the WORK table that references ARTIST.

Viewing Relationship Properties

1. Right-click the **WORK** table to display the shortcut menu, as shown in Figure 10B-53.
2. In the shortcut menu, click the **Edit** command to display the Edit Table table editor, shown in Figure 10B-54 with the column properties shown.
3. In Edit Table, click **Constraints** to display the WORK table primary key, foreign key, and unique constraint information, as shown in Figure 10B-55.
4. Note that the only foreign key we created for the WORK table is the foreign key on ArtistID, which links to ArtistID in the ARTIST column. The relevant property values are correct.
5. Close Edit Table.

Note that in Figure 10B-55 the ON DELETE value for the foreign key constraint is shown in the lower middle of Edit Table. The NO ACTION indicates that a deletion of an Artist with a corresponding work is disallowed because that would leave an ArtistID value in WORK without a corresponding value in ARTIST. This is the result of omitting the ON DELETE clause in the SQL statements for creating this foreign key in WORK, and it is the correct setting for this foreign key relationship. Also note that Oracle Database does not support ON UPDATE referential integrity actions, so no ON UPDATE actions are shown.

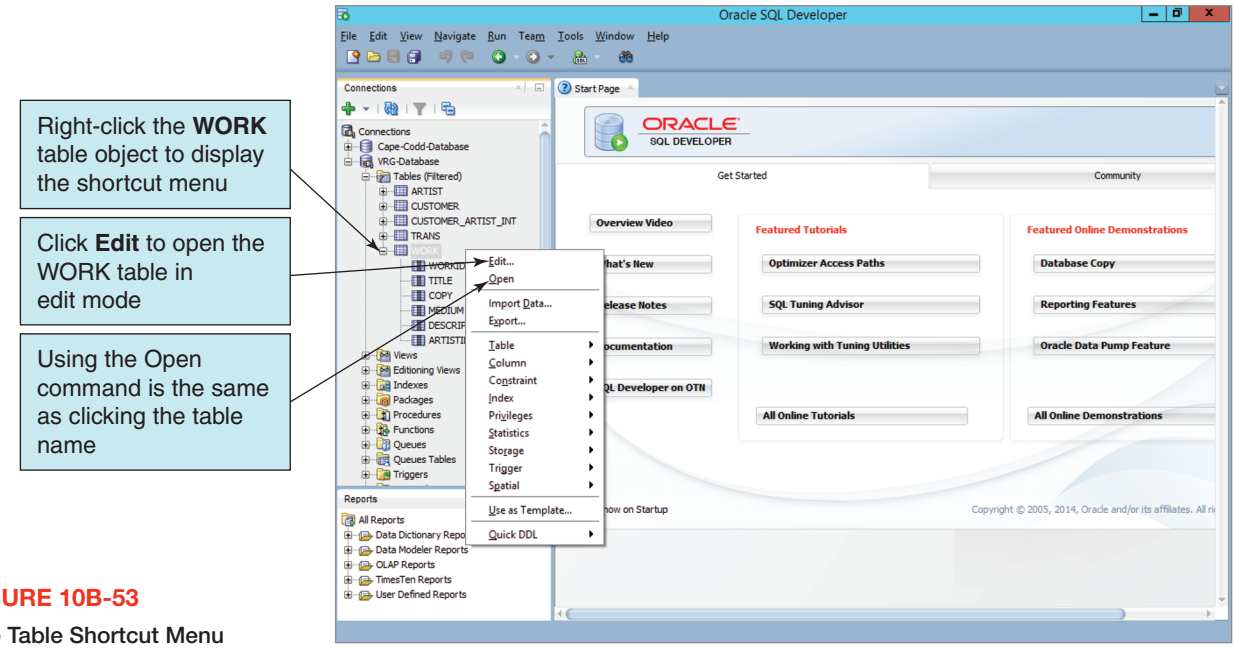


FIGURE 10B-53

The Table Shortcut Menu

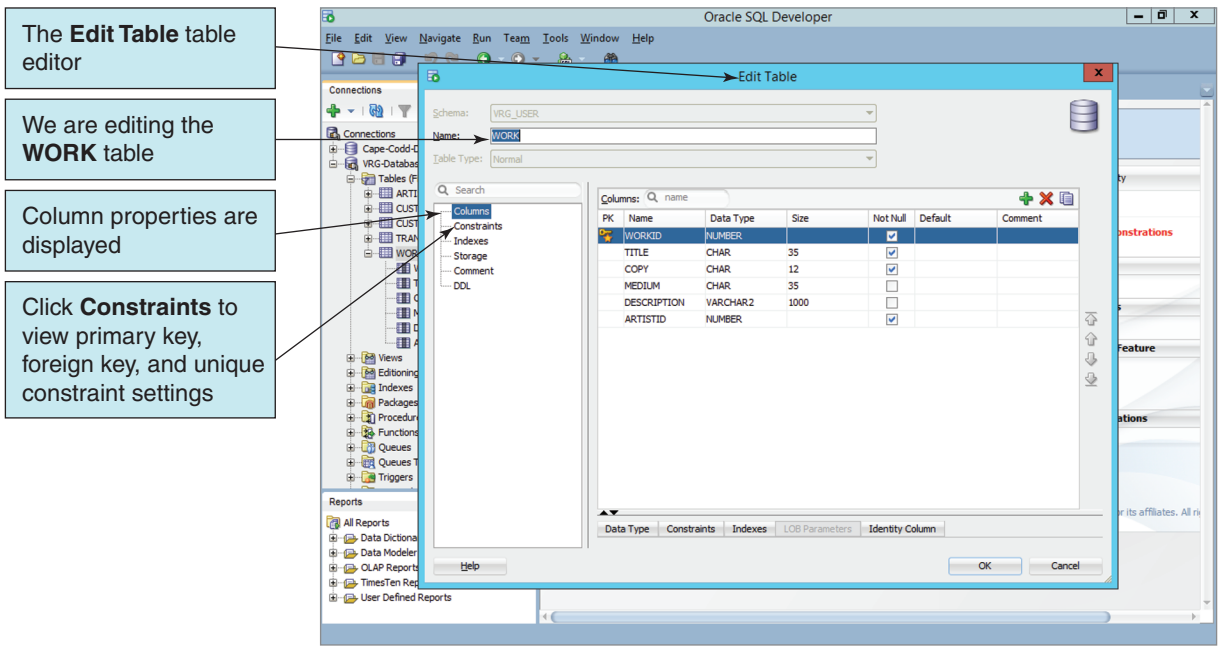
Indexes

As discussed in Appendix H, an **index** is a special data structure that is created to improve database performance. Indexes are created to enforce uniqueness on columns, to facilitate sorting, and to enable fast retrieval by column values. Columns that are frequently used with equal conditions in WHERE clauses are good candidates for indexes. The equal clause either can be a simple condition in a WHERE clause or can occur in a join. Both are shown in the following two statements:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Query-CH10B-01 *** */
SELECT      *
FROM        MYTABLE
WHERE       Column1 = 100;
```

FIGURE 10B-54

The Edit Table Table Editor



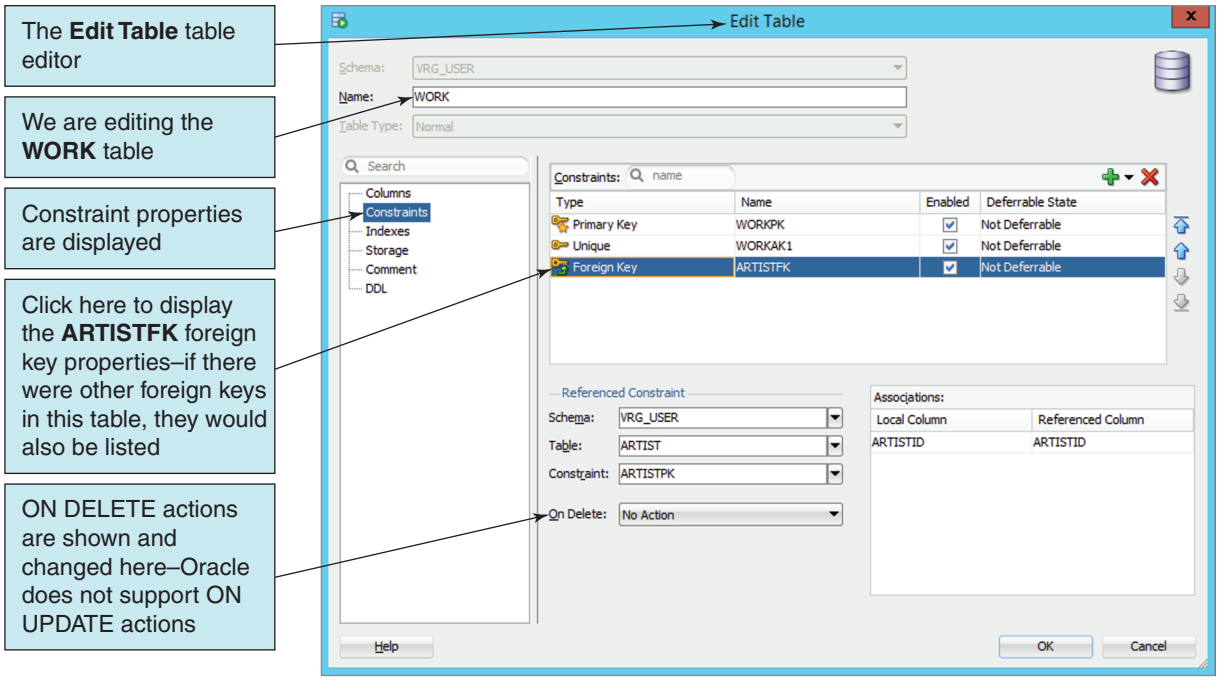


FIGURE 10B-55

The WORK Table Constraints

and

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
/* *** SQL-Query-CH10B-02 *** */
SELECT      *
FROM        MYTABLE1, MYTABLE2
WHERE       MYTABLE1.Column1 = MYTABLE2.Column2;
```

If statements like these are frequently executed, Column1 and Column2 are good candidates for indexes. For more information on Oracle Database indexes, see "Overview of Indexes" at *http://docs.oracle.com/database/121/CNCPT/indexiot.htm#CNCPT811*.

We can add an index to the CUSTOMER table on ZIPorPostalCode using the Edit Table table editor.

Creating a New Index

1. Right-click the **CUSTOMER** table to display the shortcut menu.
2. In the shortcut menu, click the **Edit** command to display the Edit Table table editor.
3. In Edit Table, click **Indexes** to display the CUSTOMER table index information, as shown in Figure 10B-56.
4. Click the **Add Index** button. Oracle Database supplies default, but incorrect, index data, as shown in Figure 10B-57.
5. Type the correct index name–*ZIPorPostalCodeIndex*–into the Index Properties Name text box. The corresponding name is immediately updated in the Indexes list.
6. Select the **ZIPorPostalCode** column name in the Expressions list by using the **Add Column Expression** and **Remove Column Expression** buttons to add the correct column to the index (this requires adding and then removing several column names). Alternately, use the drop-down arrow in the expression column.
7. The correctly specified index now appears, as shown in Figure 10B-58.
8. Click the **OK** button in the Edit Table to create the ZIPorPostalCodeIndex and close Table Edit.

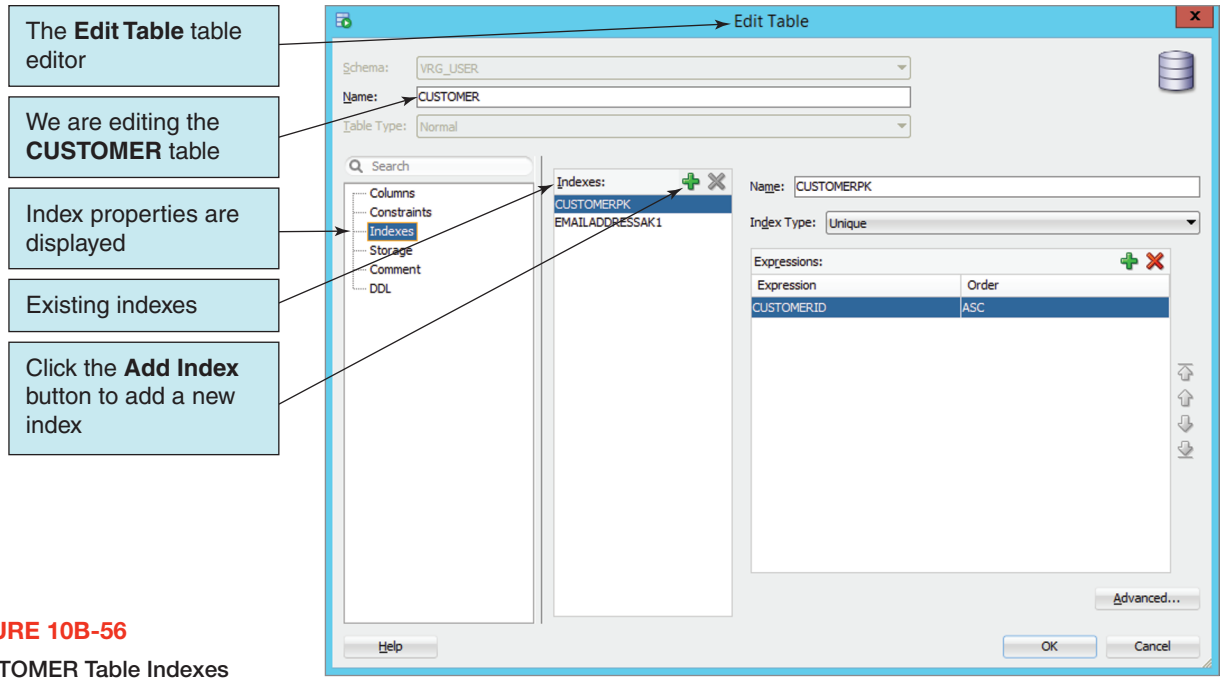


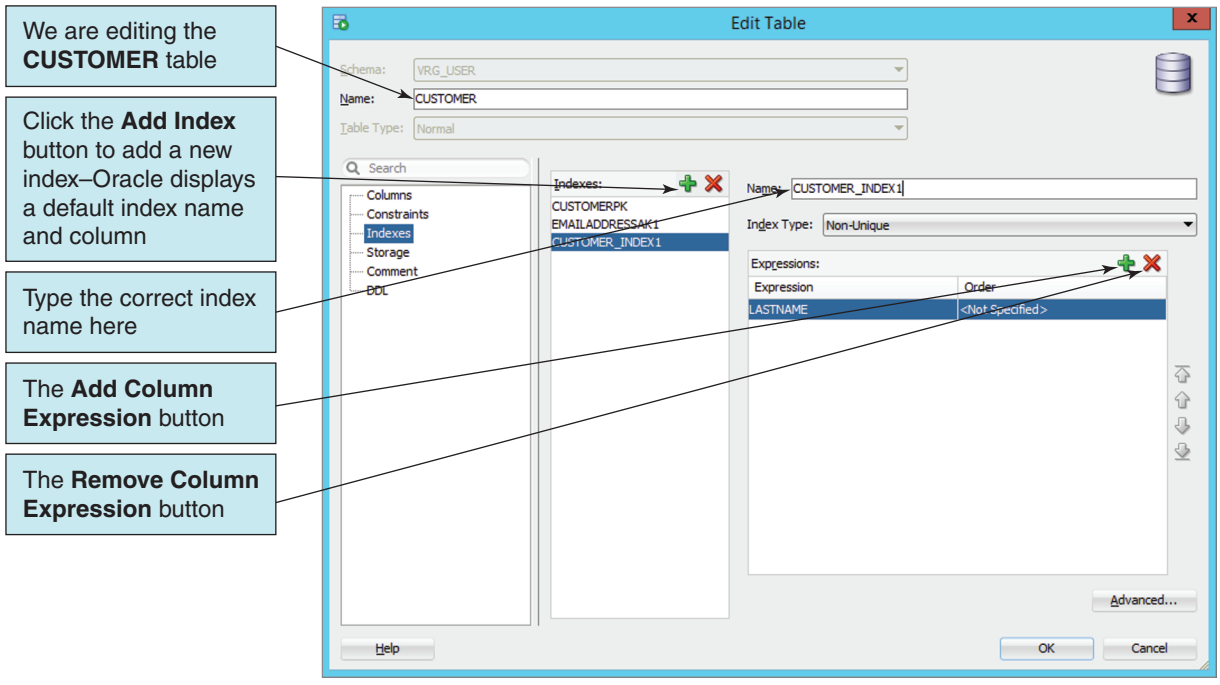
FIGURE 10B-56
CUSTOMER Table Indexes

Populating the VRG Tables

You can enter data into SQL Server either by entering data into a table grid in the SQL Developer GUI display or by using SQL INSERT statements. The SQL Developer GUI display is more useful for occasional data edits than for populating all the tables of a new database. You can open a table grid for data entry by opening the table in SQL Developer and then clicking the Data tab. This tab has an Insert Row button that enables you to add a new row if you need to. However, we will use the same method for populating the VRG database tables that we used to create the table structure: an SQL script.

Before we do that, however, we need to discuss how surrogate keys are handled in Oracle Database and address the issue of nonsequential surrogate key values that was raised in

FIGURE 10B-57
Creating a New Index



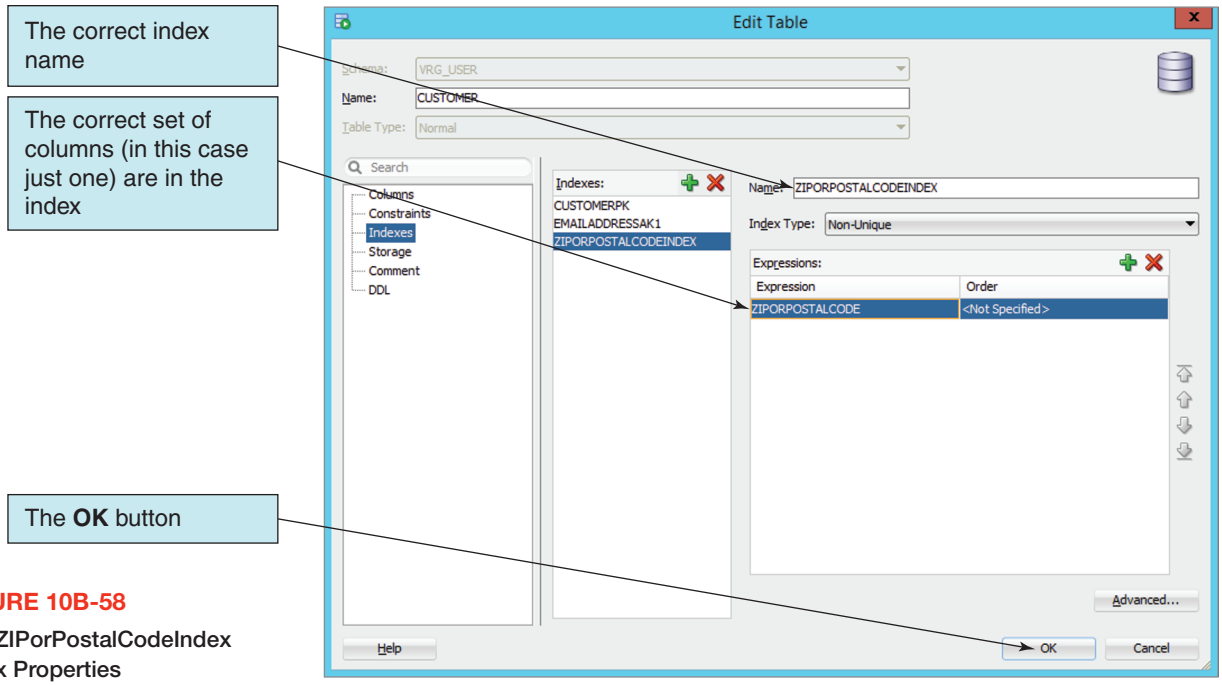


FIGURE 10B-58
The ZIPorPostalCodeIndex Index Properties

Chapter 7. The data shown in Figure 7-15 are sample data, and the primary key values of CustomerID, ArtistID, WorkID, and TransactionID shown in that figure are nonsequential.

A sequence is an Oracle Database supplied object that generates a sequential series of unique numbers. The following statement defines a sequence called seqAID that starts at 1 and is incremented by 1 each time it is used:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Sequence-10B-01 *** */
Create Sequence seqAID Increment by 1 start with 1;
```

Two sequence methods are important to us. The method NextVal provides the next value in a sequence, and the method CurrVal provides the current value in a sequence. Thus, seqAID.NextVal provides the next value of the seqAID sequence. You can insert a row into ARTIST using a sequence as follows:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-INSERT-CH10B-01 *** */
INSERT INTO ARTIST (ArtistID, LastName, FirstName, Nationality)
     VALUES (seqAID.NextVal, 'Miro', 'Joan', 'Spanish');
```

An ARTIST row will be created with the next value in the sequence as the value for ArtistID. Once this statement has been executed, you can retrieve the row just created with the CurrVal method as follows:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Query-CH10B-03 *** */
SELECT       *
FROM         ARTIST
WHERE        ArtistID = seqAID.CurrVal;
```

Here seqAID.CurrVal returns the current value of the sequence, which is the value just used.

Unfortunately, using sequences for surrogate keys has three problems. First, sequences can be used for purposes other than surrogate keys. Every time NextVal is called, a number is used up.

If the value returned from NextVal is not used for an insert into a surrogate key column but is used for something else, then that value will be missing from the surrogate key range. A second, more serious problem is that there is nothing in the schema that prevents someone from issuing an INSERT statement that does not use the sequence. Thus, Oracle Database accepts the following:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-INSERT-CH10B-02 *** */
INSERT INTO ARTIST (ArtistID, LastName, FirstName, Nationality)
     VALUES (123, 'Miro', 'Joan', 'Spanish');
```

If this were done, duplicate values of a surrogate key could occur. Third, it is possible that someone could accidentally use the wrong sequence when inserting into the table. If that were done, odd, erroneous, or duplicate surrogate key values would result.

In spite of these possible problems, sequences are the recommended way for obtaining surrogate key values in Oracle Database. At first glance, we could use the following sequences in the View Ridge database.

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Sequence-10B-02 *** */
Create Sequence seqCID Increment by 1 start with 1000;
Create Sequence seqAID Increment by 1 start with 1;
Create Sequence seqWID Increment by 1 start with 500;
Create Sequence seqTID Increment by 1 start with 100;
```

However, we still have the nonsequential surrogate key values for the VRG database data shown in Figure 7-15 to deal with. This means that if we write and execute the SQL INSERT statements to put the artist data shown in Figure 7-15(b) into the ARTIST table with the seqAID shown above in place, the values of ArtistID that will be added to the table will be (1, 2, 3, 4, 5, 6, 7, 8, 9) instead of the values of (1, 2, 3, 4, 5, 11, 17, 18, 19) listed in the figure. How can we enter the needed nonsequential values?

Fortunately, sequences give us an easy solution. We will simply insert the data in Figure 7-15 *before* we create the sequences and then create the sequences starting at the first surrogate key value we will need to use at that point.

One more point needs to be discussed. Entering values for Date data types can be problematic when using Oracle Database. Oracle Database wants dates in a particular format, but it is sometimes difficult to determine which format it wants. The **Oracle Database TO_DATE function** can be advantageous in such circumstances. TO_DATE takes two parameters, as shown here:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
TO_DATE('11/12/2015', 'MM/DD/YYYY')
```

The first parameter is the date value, and the second is the pattern to be used when interpreting the date. In this example, 11 is the month and 12 is the day of the month.

You can use the TO_DATE function with the INSERT statement to provide date values for new rows. For example, suppose that table T1 has two columns—A and B—where A is an integer and B is a date; the following insert statement can be used:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-INSERT-CH10B-03 *** */
INSERT INTO T1 VALUES (100, TO_DATE ('01/05/15', 'DD/MM/YY');
```

The result will be a new row with the values 100 and the Oracle internal format for May 1, 2015. TO_DATE can also be used with UPDATE statements.

The set of SQL INSERT statements needed to populate the VRG database with the View Ridge Gallery data shown in Figure 7-15 is shown in Figure 10B-59. Note the COMMIT

statement following the INSERT statements–this is the explicit COMMIT needed to finalize the data existence in the database.

Create and save a new SQL script named *VRG-Table-Data.sql* based on Figure 10B-59. Save the corrected script, and then run the script (use the Execute button) to populate the tables. Close the script window after the script has been successfully run.

```
/*****************************************************************************/
/*                                                                           */
/*     Kroenke and Auer - Database Processing (14th Edition) Chapter 10B     */
/*                                                                           */
/*     The View Ridge Gallery (VRG) Database - Insert Data                   */
/*                                                                           */
/*     These are the Oracle Database SQL code solutions                      */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*     This file contains the initial data for each table.                   */
/*                                                                           */
/*     We will create the surrogate key sequences after we have inserted     */
/*     this data                                                             */
/*                                                                           */
/*****************************************************************************/

/*     INSERT data for CUSTOMER                                              */


INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1000, 'Janes', 'Jeffrey', 'Jeffrey.Janes@somewhere.com', 'nh98tr3m',
       '123 W. Elm St', 'Renton', 'WA', '98055', 'USA', '425', '543-2345');

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1001, 'Smith', 'David', 'David.Smith@somewhere.com', 'ty7r932x',
       '813 Tumbleweed Lane', 'Loveland', 'CO', '81201', 'USA', '970', '654-9876');

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1015, 'Twilight', 'Tiffany', 'Tiffany.Twilight@somewhere.com',
       '88 1st Avenue', 'Langley', 'WA', '98260', 'USA', '360', '765-5566');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1033, 'Smathers', 'Fred', 'Fred.Smathers@somewhere.com',
       '10899 88th Ave', 'Bainbridge Island', 'WA', '98110', 'USA', '206', '876-9911');

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1034, 'Frederickson', 'Mary Beth', 'MaryBeth.Frederickson@somewhere.com', 'xc4vgh87',
       '25 South Lafayette', 'Denver', 'CO', '80201', 'USA', '303', '513-8822');
```

FIGURE 10B-59

The SQL Statements to Populate the VRG Tables

```
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1036, 'Warning', 'Selma', 'Selma.Warning@somewhere.com', 'ca45b32c',
       '205 Burnaby', 'Vancouver', 'BC', 'V6Z 1W2', 'Canada', '604', '988-0512');

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1037, 'Wu', 'Susan', 'Susan.Wu@somewhere.com',
       '105 Locust Ave', 'Atlanta', 'GA', '30322', 'USA', '404', '653-3465');

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1040, 'Gray', 'Donald','Donald.Gray@somewhere.com', '98zx3y6',
       '55 Bodega Ave', 'Bodega Bay', 'CA', '94923', 'USA', '707', '568-4839');

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1041, 'Johnson', 'Lynda',
       '117 C Street', 'Washington', 'DC', '20003', 'USA', '202', '438-5498');

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, EmailAddress, EncryptedPassword,
        Street, City, State, ZIPorPostalCode, Country, AreaCode, PhoneNumber)
       VALUES (
       1051, 'Wilkens', 'Chris', 'Chris.Wilkens@somewhere.com', 'wqpb3yyu',
       '87 Highland Drive', 'Olympia', 'WA', '98508', 'USA', '360', '876-8822');

/*     CREATE the CUSTOMER surrogate key sequence seqCID                      */

CREATE SEQUENCE seqCID INCREMENT BY 1 START WITH 1052;

/*****************************************************************************/

/*     INSERT data for ARTIST                                                 */

INSERT INTO ARTIST VALUES(
       1, 'Miro', 'Joan', 'Spanish', 1893, 1983);
INSERT INTO ARTIST VALUES(
       2, 'Kandinsky', 'Wassily', 'Russian', 1866, 1944);
INSERT INTO ARTIST VALUES(
       3, 'Klee', 'Paul', 'German', 1879, 1940);
INSERT INTO ARTIST VALUES(
       4, 'Matisse', 'Henri', 'French', 1869, 1954);
INSERT INTO ARTIST VALUES(
       5, 'Chagall', 'Marc', 'French', 1887, 1985);
INSERT INTO ARTIST VALUES(
       11, 'Sargent', 'John Singer', 'United States', 1856, 1925);
INSERT INTO ARTIST VALUES(
       17, 'Tobey', 'Mark', 'United States', 1890, 1976);
INSERT INTO ARTIST VALUES(
       18, 'Horiuchi', 'Paul', 'United States', 1906, 1999);
INSERT INTO ARTIST VALUES(
       19, 'Graves', 'Morris', 'United States', 1920, 2001);

/*     CREATE the CUSTOMER surrogate key sequence seqCID                      */

CREATE SEQUENCE seqAID INCREMENT BY 1 START WITH 20;

/*****************************************************************************/
```

FIGURE 10B-59 (*continued*)

Continued

```
/*     INSERT data for CUSTOMER_ARTIST_INT                                    */

INSERT INTO CUSTOMER_ARTIST_INT VALUES (1, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (1, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (2, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (2, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (4, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (4, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1051);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1051);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1051);

/*****************************************************************************/

/*     INSERT data for WORK                                                   */

INSERT INTO WORK VALUES(
       500, 'Memories IV', 'Unique', 'Casein rice paper collage', '31 x 24.8 in.', 18);
INSERT INTO WORK VALUES(
       511, 'Surf and Bird', '142/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK VALUES(
       521, 'The Tilled Field', '788/1000', 'High Quality Limited Print',
       'Early Surrealist style', 1);
INSERT INTO WORK VALUES(
       522, 'La Lecon de Ski', '353/500', 'High Quality Limited Print',
       'Surrealist style', 1);
INSERT INTO WORK VALUES(
       523, 'On White II', '435/500', 'High Quality Limited Print',
       'Bauhaus style of Kandinsky', 2);
INSERT INTO WORK VALUES(
       524, 'Woman with a Hat', '596/750', 'High Quality Limited Print',
       'A very colorful Impressionist piece', 4);
INSERT INTO WORK VALUES(
       537, 'The Woven World', '17/750', 'Color lithograph', 'Signed', 17);
INSERT INTO WORK VALUES(
       548, 'Night Bird', 'Unique', 'Watercolor on Paper',
       '50 x 72.5 cm. - Signed', 19);
INSERT INTO WORK VALUES(
       551, 'Der Blaue Reiter', '236/1000', 'High Quality Limited Print',
       'The Blue Rider-Early Pointilism influence', 2);
INSERT INTO WORK VALUES(
       552, 'Angelus Novus', '659/750', 'High Quality Limited Print',
       'Bauhaus style of Klee', 3);
```

FIGURE 10B-59
Continued

```
INSERT INTO WORK VALUES(
       553, 'The Dance', '734/1000', 'High Quality Limited Print',
       'An Impressionist masterpiece', 4);
INSERT INTO WORK VALUES(
       554, 'I and the Village', '834/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK VALUES(
       555, 'Claude Monet Painting', '684/1000', 'High Quality Limited Print',
       'Shows French Impressionist influence of Monet', 11);
INSERT INTO WORK VALUES(
       561, 'Sunflower', 'Unique', 'Watercolor and ink',
       '33.3 x 16.1 cm. - Signed', 19);
INSERT INTO WORK VALUES(
       562, 'The Fiddler', '251/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK VALUES(
       563, 'Spanish Dancer', '583/750', 'High Quality Limited Print',
       'American realist style - From work in Spain', 11);
INSERT INTO WORK VALUES(
       564, 'Farmer''s Market #2',   '267/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK VALUES(
       565, 'Farmer''s Market #2',   '268/500', 'High Quality Limited Print',
        'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK VALUES(
       566, 'Into Time', '323/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 18);
INSERT INTO WORK VALUES(
       570, 'Untitled Number 1', 'Unique', 'Monotype with tempera',
       '4.3 x 6.1 in. Signed', 17);
INSERT INTO WORK VALUES(
       571, 'Yellow Covers Blue', 'Unique', 'Oil and collage',
       '71 x 78 in. - Signed', 18);
INSERT INTO WORK VALUES(
       578, 'Mid-Century Hibernation', '362/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK VALUES(
       580, 'Forms in Progress I', 'Unique', 'Color aquatint',
       '19.3 x 24.4 in. - Signed', 17);
INSERT INTO WORK VALUES(
       581, 'Forms in Progress II', 'Unique', 'Color aquatint',
       '19.3 x 24.4 in. - Signed', 17);
INSERT INTO WORK VALUES(
       585, 'The Fiddler', '252/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK VALUES(
       586, 'Spanish Dancer', '588/750', 'High Quality Limited Print',
       'American Realist style - From work in Spain', 11);
INSERT INTO WORK VALUES(
       587, 'Broadway Boggie', '433/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK VALUES(
       588, 'Universal Field', '114/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK VALUES(
       589, 'Color Floating in Time', '487/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 18);
INSERT INTO WORK VALUES(
       590, 'Blue Interior', 'Unique', 'Tempera on card', '43.9 x 28 in.', 17);
INSERT INTO WORK VALUES(
       593, 'Surf and Bird', 'Unique', 'Gouache', '26.5 x 29.75 in. - Signed', 19);
INSERT INTO WORK VALUES(
       594, 'Surf and Bird', '362/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
```

FIGURE 10B-59 (*continued*)

Continued

```
INSERT INTO WORK VALUES(
       595, 'Surf and Bird', '365/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK VALUES(
       596, 'Surf and Bird', '366/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);

/*     CREATE the WORK surrogate key sequence seqWID                          */

CREATE SEQUENCE seqWID INCREMENT BY 1 START WITH 597;

/*****************************************************************************/

/*     INSERT data for TRANS                                                  */

INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       100, TO_DATE('11/04/2011','MM/DD/YYYY'), 30000.00, 45000.00,
       TO_DATE('12/14/2011','MM/DD/YYYY'), 42500.00, 1000, 500);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       101, TO_DATE('11/07/2011','MM/DD/YYYY'), 250.00, 500.00,
       TO_DATE('12/19/2011','MM/DD/YYYY'), 500.00, 1015, 511);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       102, TO_DATE('11/17/2011','MM/DD/YYYY'), 125.00, 250.00,
       TO_DATE('01/18/2012','MM/DD/YYYY'), 200.00, 1001, 521);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       103, TO_DATE('11/17/2011','MM/DD/YYYY'), 250.00, 500.00,
       TO_DATE('12/12/2012','MM/DD/YYYY'), 400.00, 1034, 522);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       104, TO_DATE('11/17/2011','MM/DD/YYYY'), 250.00, 250.00,
       TO_DATE('01/18/2012','MM/DD/YYYY'), 200.00, 1001, 523);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       105, TO_DATE('11/17/2011','MM/DD/YYYY'), 200.00, 500.00,
       TO_DATE('12/12/2012','MM/DD/YYYY'), 400.00, 1034, 524);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       115, TO_DATE('03/03/2012','MM/DD/YYYY'), 1500.00, 3000.00,
       TO_DATE('06/07/2012','MM/DD/YYYY'), 2750.00, 1033, 537);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       121, TO_DATE('09/21/2012','MM/DD/YYYY'), 15000.00, 30000.00,
       TO_DATE('11/28/2012','MM/DD/YYYY'), 27500.00,  1015, 548);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       125, TO_DATE('11/21/2012','MM/DD/YYYY'), 125.00, 250.00,
       TO_DATE('12/18/2012','MM/DD/YYYY'), 200.00, 1001, 551);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       126, TO_DATE('11/21/2012','MM/DD/YYYY'), 200.00, 400.00, 552);
```

FIGURE 10B-59

Continued

```
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       127, TO_DATE('11/21/2012','MM/DD/YYYY'), 125.00, 500.00,
       TO_DATE('12/22/2012','MM/DD/YYYY'), 400.00, 1034, 553);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       128, TO_DATE('11/21/2012','MM/DD/YYYY'), 125.00, 250.00,
       TO_DATE('03/16/2013','MM/DD/YYYY'), 225.00, 1036, 554);

INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       129, TO_DATE('11/21/2012','MM/DD/YYYY'), 125.00, 250.00,
       TO_DATE('03/16/2013','MM/DD/YYYY'), 225.00, 1036, 555);

INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       151, TO_DATE('05/07/2013','MM/DD/YYYY'), 10000.00, 20000.00,
       TO_DATE('06/28/2013','MM/DD/YYYY'), 17500.00, 1036, 561);

INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       152, TO_DATE('05/18/2013','MM/DD/YYYY'), 125.00, 250.00,
       TO_DATE('08/15/2013','MM/DD/YYYY'), 225.00, 1001, 562);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       153, TO_DATE('05/18/2013','MM/DD/YYYY'), 200.00, 400.00,
       TO_DATE('08/15/2013','MM/DD/YYYY'), 350.00, 1001, 563);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       154, TO_DATE('05/18/2013','MM/DD/YYYY'), 250.00, 500.00,
       TO_DATE('09/28/2013','MM/DD/YYYY'), 400.00, 1040, 564);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       155, TO_DATE('05/18/2013','MM/DD/YYYY'), 250.00, 500.00, 565);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       156, TO_DATE('05/18/2013','MM/DD/YYYY'), 250.00, 500.00,
       TO_DATE('09/27/2013','MM/DD/YYYY'), 400.00, 1040, 566);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       161, TO_DATE('06/28/2013','MM/DD/YYYY'), 7500.00, 15000.00,
       TO_DATE('09/29/2013','MM/DD/YYYY'), 13750.00, 1033, 570);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       171, TO_DATE('08/23/2013','MM/DD/YYYY'), 35000.00, 60000.00,
       TO_DATE('09/29/2013','MM/DD/YYYY'), 55000.00, 1000, 571);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       175, TO_DATE('09/29/2013','MM/DD/YYYY'), 40000.00, 75000.00,
       TO_DATE('12/18/2013','MM/DD/YYYY'), 72500.00, 1036, 500);
```

FIGURE 10B-59 (*continued*)

Continued

```
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       181, TO_DATE('10/11/2013','MM/DD/YYYY'), 250.00, 500.00, 578);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       201, TO_DATE('02/28/2014','MM/DD/YYYY'), 2000.00, 3500.00,
       TO_DATE('04/26/2014','MM/DD/YYYY'), 3250.00,   1040, 580);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       202, TO_DATE('02/28/2014','MM/DD/YYYY'), 2000.00, 3500.00,
       TO_DATE('04/26/2014','MM/DD/YYYY'), 3250.00, 1040, 581);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       225, TO_DATE('06/08/2014','MM/DD/YYYY'), 125.00, 250.00,
       TO_DATE('09/27/2014','MM/DD/YYYY'), 225.00, 1051, 585);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       226, TO_DATE('06/08/2014','MM/DD/YYYY'), 200.00, 400.00, 586);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       227, TO_DATE('06/08/2014','MM/DD/YYYY'), 250.00, 500.00,
       TO_DATE('09/27/2014','MM/DD/YYYY'), 475.00, 1051, 587);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       228, TO_DATE('06/08/2014','MM/DD/YYYY'), 250.00, 500.00, 588);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       229, TO_DATE('06/08/2014','MM/DD/YYYY'), 250.00, 500.00, 589);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       241, TO_DATE('08/29/2014','MM/DD/YYYY'), 2500.00, 5000.00,
       TO_DATE('09/27/2014','MM/DD/YYYY'), 4750.00,   1015, 590);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       251, TO_DATE('10/25/2014','MM/DD/YYYY'), 25000.00, 50000.00, 593);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       252, TO_DATE('10/27/2014','MM/DD/YYYY'), 250.00, 500.00, 594);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       253, TO_DATE('10/27/2014','MM/DD/YYYY'), 250.00, 500.00, 595);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       254, TO_DATE('10/27/2014','MM/DD/YYYY'), 250.00, 500.00, 596);

/*     CREATE the TRANS surrogate key sequence seqTID                    */

CREATE SEQUENCE seqTID INCREMENT BY 1 START WITH 255;

/*     COMMIT the transaction                                            */

COMMIT;
/*************************************************************************/
```

FIGURE 10B-59

Continued

Creating Views

SQL views were discussed in Chapter 7. One view we created in that discussion was *CustomerInterestsView*. Views can be created with an SQL statement or in the GUI display by right-clicking the Views folder to display a shortcut menu and then clicking the New View command. However, the SQL Developer GUI tool basically gives us an SQL template, so we might as well just use SQL in an SQL Worksheet window. This view can be created with the SQL statement:

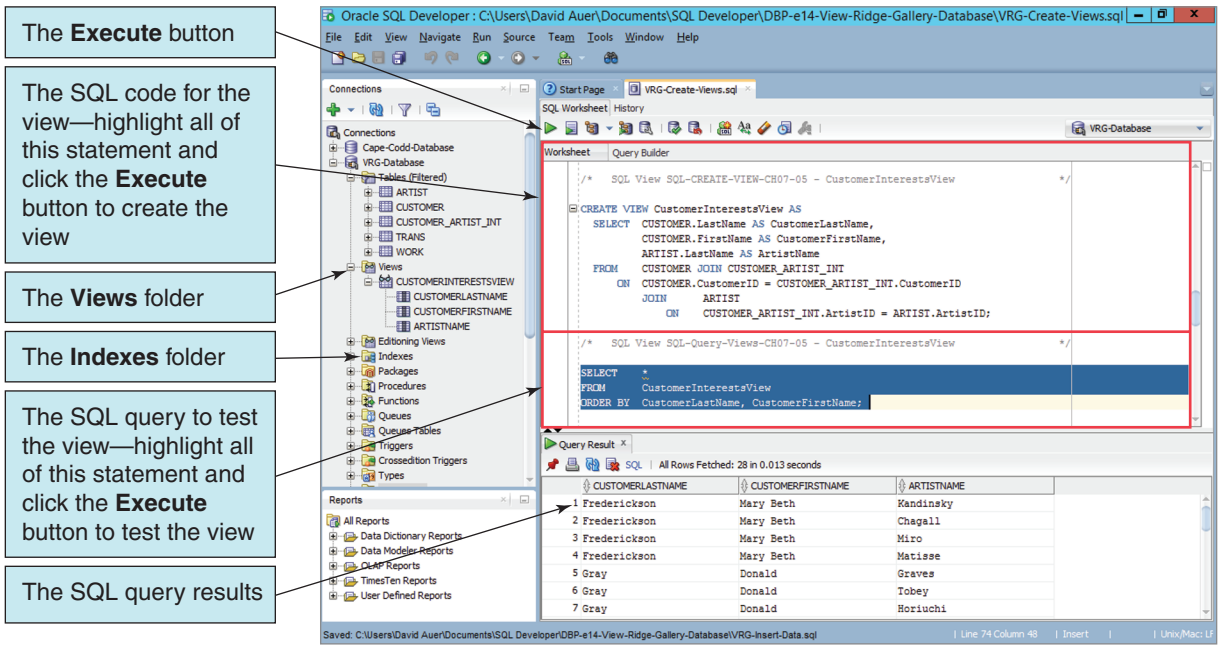
```
/* *** SQL-CREATE-VIEW-CH07-05 *** */
CREATE VIEW CustomerInterestsView AS
    SELECT      CUSTOMER.LastName AS CustomerLastName,
                CUSTOMER.FirstName AS CustomerFirstName,
                ARTIST.LastName AS ArtistName
    FROM        CUSTOMER JOIN CUSTOMER_ARTIST_INT
        ON      CUSTOMER.CustomerID =
                CUSTOMER_ARTIST_INT.CustomerID
                JOIN      ARTIST
                    ON    CUSTOMER_ARTIST_INT.ArtistID =
                          ARTIST.ArtistID;
```

Figure 10B-60 shows this SQL CREATE VIEW statement in an SQL script named *VRG-Create-Views.sql* in the SQL Developer. The VRG-Create-Views.sql script contains Oracle Database versions of all the views that we wrote in Chapter 7.

Creating a New View

1. In SQL Developer, click the **Open SQL Worksheet** button to open a new tabbed SQL Worksheet document window.
2. Type the **SQL Statements** to create the CustomerInterestsView view shown in Figure 10B-60.
3. Click the **Execute** button.
4. To save this CREATE VIEW statement as part of an SQL script, save the script as **VRG-Create-Views.sql**.
5. Close the SQL Worksheet.

FIGURE 10B-60
Creating an SQL View



As explained in Chapter 7, SQL views are used like tables in other SQL statements. For example, to see all the data in the view, we use the SQL SELECT statement:

```
/* *** SQL-SELECT-VIEW-CH07-05 *** */
SELECT          *
FROM            CustomerInterestsView
ORDER BY        CustomerLastName, CustomerFirstName;
```

The result is shown in Figure 10B-61. At this point, you should create all the views discussed in Chapter 7 and add them to your VRG-Create-Views.sql script. For each view, include the SQL statement needed to test the view.

BY THE WAY Oracle Database allows the ORDER BY clause in view definitions.

FIGURE 10B-61
Result of Using the View CustomerInterestsView

	CUSTOMERLASTNAME	CUSTOMERFIRSTNAME	ARTISTNAME
1	Frederickson	Mary Beth	Kandinsky
2	Frederickson	Mary Beth	Chagall
3	Frederickson	Mary Beth	Miro
4	Frederickson	Mary Beth	Matisse
5	Gray	Donald	Graves
6	Gray	Donald	Tobey
7	Gray	Donald	Horiuchi
8	Janes	Jeffrey	Horiuchi
9	Janes	Jeffrey	Graves
10	Janes	Jeffrey	Tobey
11	Smathers	Fred	Horiuchi
12	Smathers	Fred	Tobey
13	Smathers	Fred	Graves
14	Smith	David	Chagall
15	Smith	David	Matisse
16	Smith	David	Miro
17	Smith	David	Kandinsky
18	Smith	David	Sargent
19	Twilight	Tiffany	Tobey
20	Twilight	Tiffany	Sargent
21	Twilight	Tiffany	Graves
22	Twilight	Tiffany	Horiuchi
23	Warning	Selma	Chagall
24	Warning	Selma	Graves
25	Warning	Selma	Sargent
26	Wilkens	Chris	Graves
27	Wilkens	Chris	Tobey
28	Wilkens	Chris	Horiuchi

Importing Microsoft Excel Data into an Oracle Database Table

When developing a database to support an application, it is very common to find that some (if not all) of the data needed in the database exists as data in user **worksheets** (also called **spreadsheets**). A typical example of this is a Microsoft Excel 2013 worksheet that a user has been maintaining and that must now be converted to data stored in the database.

If we are really lucky, the worksheet will already be organized like a database table, with appropriate column labels and unique data in each row. And if we are *really, really lucky*, there will be one or more columns that can be used as the primary key in the new database table. In that case, we can easily import the data into the database. More likely, we will have to modify the worksheet and organize and clean up the data in it before we can import the data. In essence, we are following a procedure that we will encounter again in Chapter 12 in our discussion of data warehouses known as **extract, transform, and load (ETL)**.

As an example, let's consider the problem of postcards sold by the View Ridge Gallery. These postcards are pictures of the artwork sold by the View Ridge Gallery (postcards of other popular works of art by well-known artists such as Claude Monet's *Water Lilies* are also stocked and sold, but to simplify the problem, we will consider only postcards of artwork at the View Ridge Gallery). The inventory of postcards is currently kept in a Microsoft Excel 2013 worksheet named POSTCARDS and shown in Figure 10B-62(a).

We will need a primary key in our Oracle Database table, and, unfortunately, the Oracle Data Import Wizard that we will use does not allow us to add a primary key during the data import process. Therefore, in Microsoft Excel 2013 we will make a copy of the POSTCARDS worksheet named POSTCARDSwithID and manually add a *PostcardID* identifier column, as shown in Figure 10B-62(b).

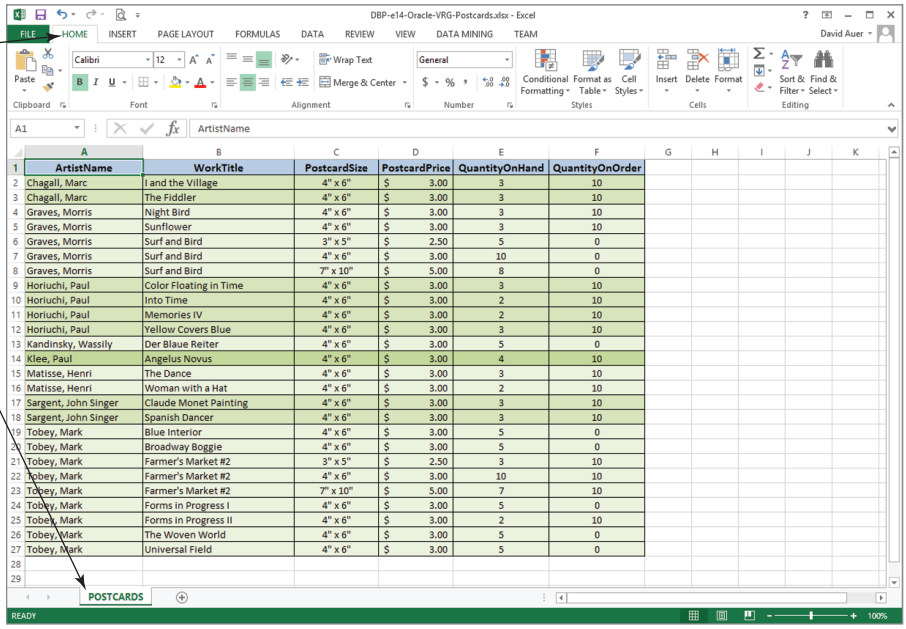
Even with the added PostcardID column, this worksheet breaks our basic rule of one theme per table and combines artist, artwork, and inventory into the same worksheet. Fortunately, there is no multivalue, multicolumn problem (as discussed in Chapter 4) in this worksheet. Even as it stands, this worksheet would need normalizing into BCNF to create the proper set of tables for the VRG database.

However, we will deal with the normalization in the VRG database itself in the Review Questions at the end of this chapter. For now, we will simply import that worksheet into the VRG database and use it as a temporary table and source of data for populating the properly normalized tables.

FIGURE 10B-62
The View Ridge Gallery POSTCARD Worksheet

The **Home** Command Tab

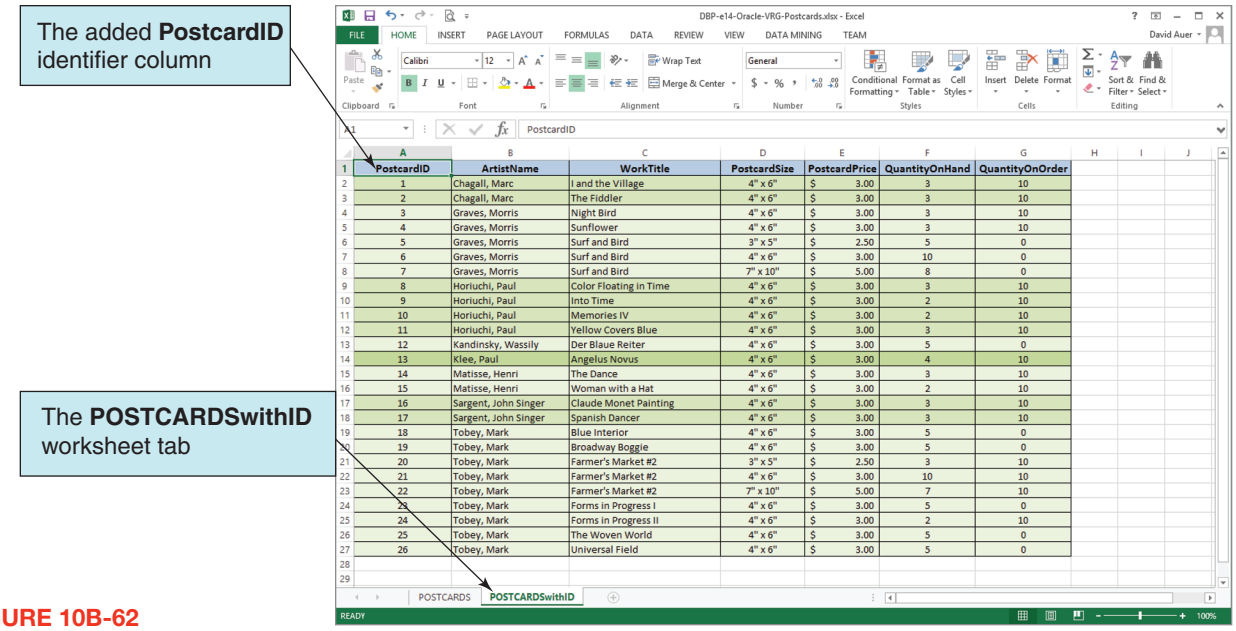
The **POSTCARDS** worksheet tab



ArtistName	WorkTitle	PostcardSize	PostcardPrice	QuantityOnHand	QuantityOnOrder
Chagall, Marc	I and the Village	4" x 6"	$ 3.00	3	10
Chagall, Marc	The Fiddler	4" x 6"	$ 3.00	3	10
Graves, Morris	Night Bird	4" x 6"	$ 3.00	3	10
Graves, Morris	Sunflower	4" x 6"	$ 3.00	3	10
Graves, Morris	Surf and Bird	3" x 5"	$ 2.50	5	0
Graves, Morris	Surf and Bird	4" x 6"	$ 3.00	10	0
Graves, Morris	Surf and Bird	7" x 10"	$ 5.00	8	0
Horiuchi, Paul	Color Floating in Time	4" x 6"	$ 3.00	3	10
Horiuchi, Paul	Into Time	4" x 6"	$ 3.00	2	10
Horiuchi, Paul	Memories IV	4" x 6"	$ 3.00	2	10
Horiuchi, Paul	Yellow Covers Blue	4" x 6"	$ 3.00	3	10
Kandinsky, Wassily	Der Blaue Reiter	4" x 6"	$ 3.00	5	0
Klee, Paul	Angelus Novus	4" x 6"	$ 3.00	4	10
Matisse, Henri	The Dance	4" x 6"	$ 3.00	3	10
Matisse, Henri	Woman with a Hat	4" x 6"	$ 3.00	2	10
Sargent, John Singer	Claude Monet Painting	4" x 6"	$ 3.00	3	10
Sargent, John Singer	Spanish Dancer	4" x 6"	$ 3.00	3	10
Tobey, Mark	Blue Interior	4" x 6"	$ 3.00	5	0
Tobey, Mark	Broadway Boggie	4" x 6"	$ 3.00	5	0
Tobey, Mark	Farmer's Market #2	3" x 5"	$ 2.50	3	10
Tobey, Mark	Farmer's Market #2	4" x 6"	$ 3.00	10	10
Tobey, Mark	Farmer's Market #2	7" x 10"	$ 5.00	7	10
Tobey, Mark	Forms in Progress I	4" x 6"	$ 3.00	5	0
Tobey, Mark	Forms in Progress II	4" x 6"	$ 3.00	2	10
Tobey, Mark	The Woven World	4" x 6"	$ 3.00	5	0
Tobey, Mark	Universal Field	4" x 6"	$ 3.00	5	0

(a) The Original Worksheet

(*continued*)



PostcardID	ArtistName	WorkTitle	PostcardSize	PostcardPrice	QuantityOnHand	QuantityOnOrder
1	Chagall, Marc	I and the Village	4" x 6"	$ 3.00	3	10
2	Chagall, Marc	The Fiddler	4" x 6"	$ 3.00	3	10
3	Graves, Morris	Night Bird	4" x 6"	$ 3.00	3	10
4	Graves, Morris	Sunflower	4" x 6"	$ 3.00	3	10
5	Graves, Morris	Surf and Bird	3" x 5"	$ 2.50	5	0
6	Graves, Morris	Surf and Bird	4" x 6"	$ 3.00	10	0
7	Graves, Morris	Surf and Bird	7" x 10"	$ 5.00	8	0
8	Horiuchi, Paul	Color Floating in Time	4" x 6"	$ 3.00	3	10
9	Horiuchi, Paul	Into Time	4" x 6"	$ 3.00	2	10
10	Horiuchi, Paul	Memories IV	4" x 6"	$ 3.00	2	10
11	Horiuchi, Paul	Yellow Covers Blue	4" x 6"	$ 3.00	3	10
12	Kandinsky, Wassily	Der Blaue Reiter	4" x 6"	$ 3.00	5	0
13	Klee, Paul	Angelus Novus	4" x 6"	$ 3.00	4	10
14	Matisse, Henri	The Dance	4" x 6"	$ 3.00	3	10
15	Matisse, Henri	Woman with a Hat	4" x 6"	$ 3.00	2	10
16	Sargent, John Singer	Claude Monet Painting	4" x 6"	$ 3.00	3	10
17	Sargent, John Singer	Spanish Dancer	4" x 6"	$ 3.00	3	10
18	Tobey, Mark	Blue Interior	4" x 6"	$ 3.00	5	0
19	Tobey, Mark	Broadway Boggie	4" x 6"	$ 3.00	5	0
20	Tobey, Mark	Farmer's Market #2	3" x 5"	$ 2.50	3	10
21	Tobey, Mark	Farmer's Market #2	4" x 6"	$ 3.00	10	10
22	Tobey, Mark	Farmer's Market #2	7" x 10"	$ 5.00	7	10
23	Tobey, Mark	Forms in Progress I	4" x 6"	$ 3.00	5	0
24	Tobey, Mark	Forms in Progress II	4" x 6"	$ 3.00	2	10
25	Tobey, Mark	The Woven World	4" x 6"	$ 3.00	5	0
26	Tobey, Mark	Universal Field	4" x 6"	$ 3.00	5	0

FIGURE 10B-62
Continued

(b) The Copied Worksheet with Postcard ID Column

Oracle Database provides two ways of importing Microsoft Excel data via SQL Developer:

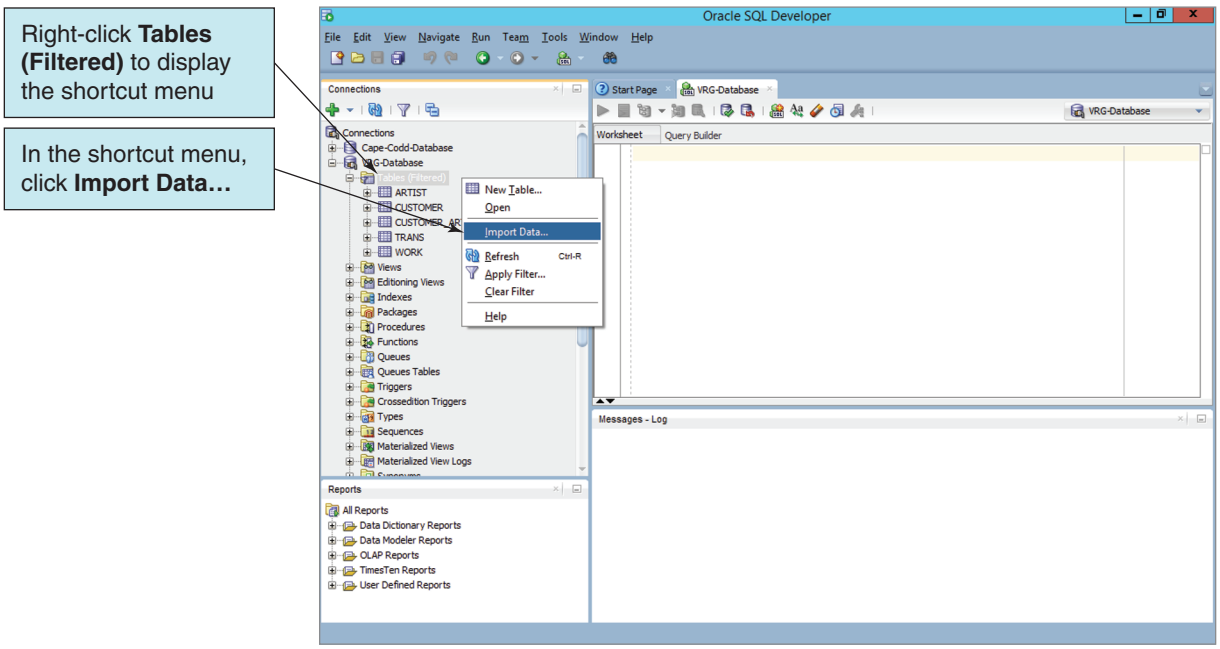
- Create the table first using an SQL CREATE TABLE statement, and then import the data.
- Create the table while importing the data.

We will use the second method.

Importing the Microsoft Excel Data into an Oracle Database Database Table

1. In Oracle SQL Developer, expand the VRG database.
2. Right-click on the Tables (Filtered) VRG database object to display a shortcut menu, and in the shortcut menu click on the **Import Data** command, as shown in Figure 10B-63.

FIGURE 10B-63
The Import Data Command



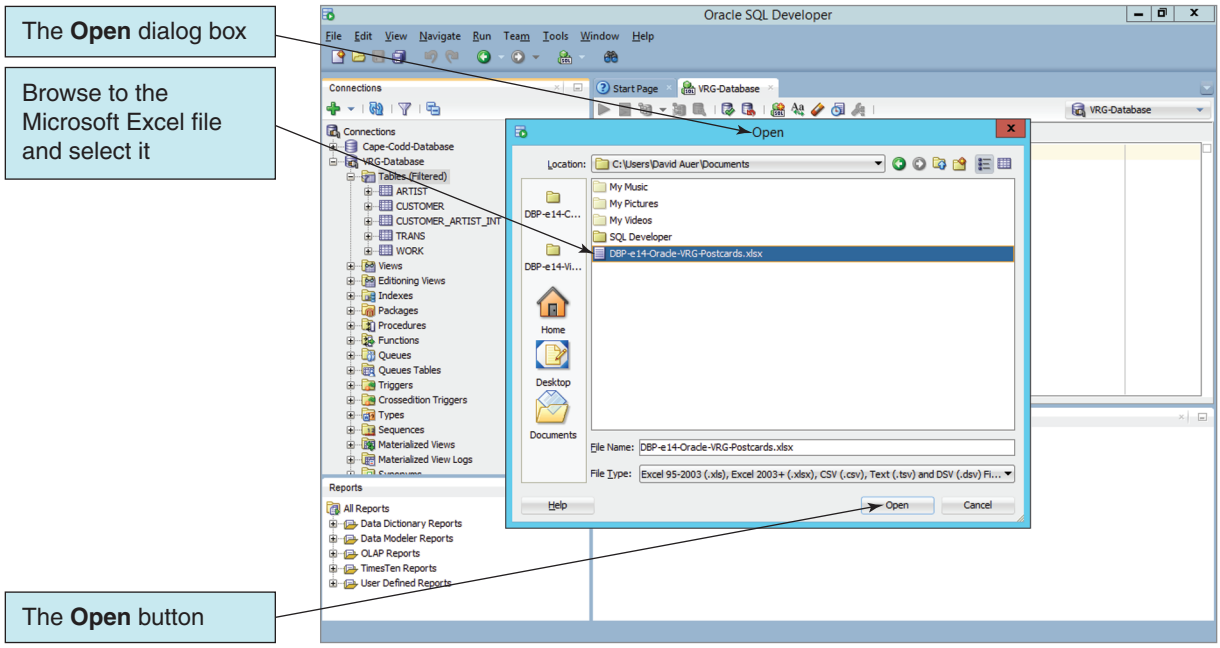


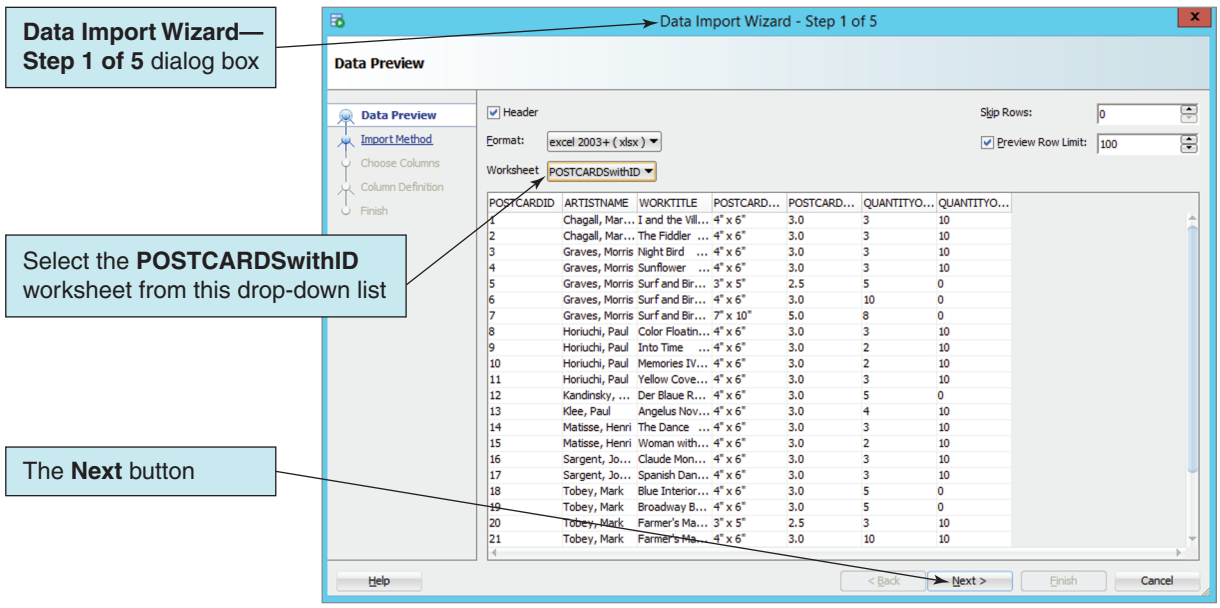
FIGURE 10B-64

The Open Dialog Box for the Excel Workbook File

3. Click the **Import Data** command shown in Figure 10B-63. The Open dialog box is displayed, as shown in Figure 10B-64. Browse to the Excel workbook, and then click the **Open** button.
4. The *Data Import Wizard–Step 1 of 5* (Data Preview) dialog box is displayed, shown in Figure 10B-65. Select the **POSTCARDSwithID** worksheet from the Worksheet drop-down list as shown in the figure. Also, as shown in the figure, make sure the Header checkbox is checked and that the **Excel 2003+ (xlsx)** format is selected from the Format drop-down list.
5. Click the **Next** button. The *Data Import Wizard–Step 2 of 5* (Import Data) dialog box is displayed (and mislabeled as *Step 2 of 4*). Type in the Table Name **POSTCARDS_TEMP** so that the dialog box appears as shown in Figure 10B-66. The rest of the settings are correct.

FIGURE 10B-65

The Data Import Wizard—Step 1 of 5 Dialog Box



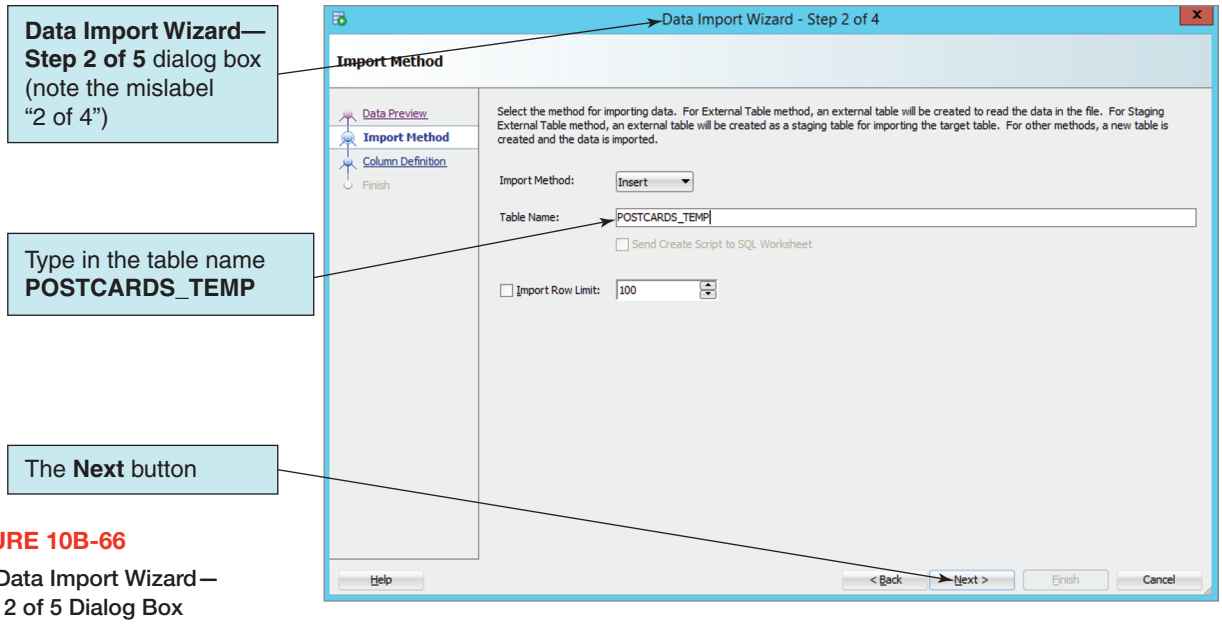
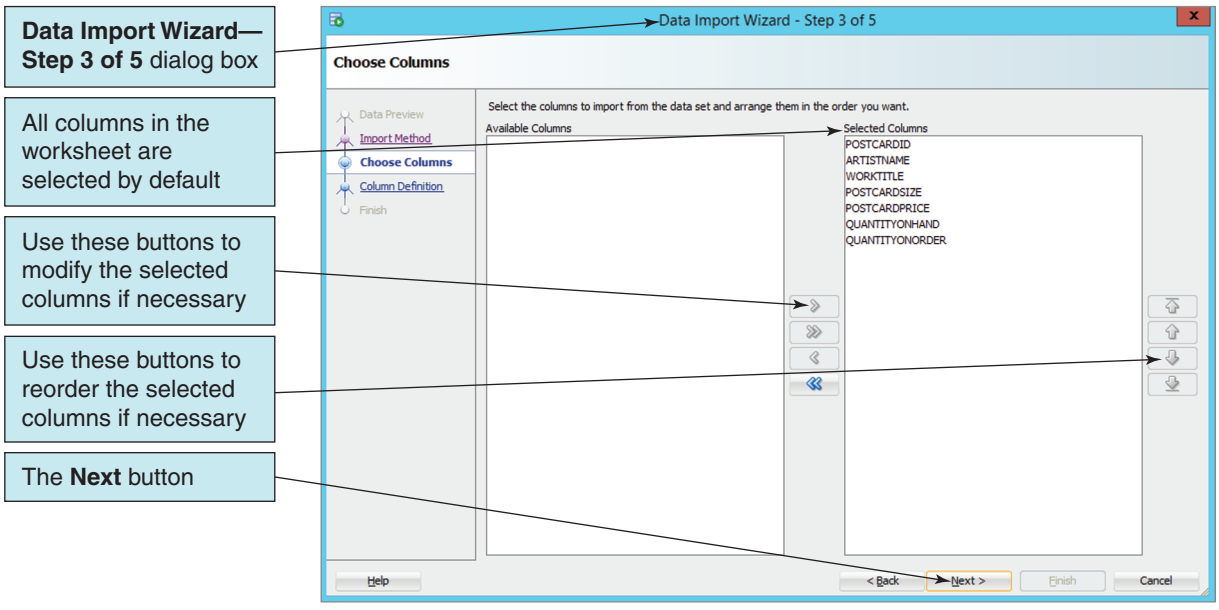


FIGURE 10B-66
The Data Import Wizard—Step 2 of 5 Dialog Box

6. Click the **Next** button. The *Data Import Wizard–Step 3 of 5* dialog box is displayed as shown in Figure 10B-67. This step allows us to choose which worksheet columns to import. Note that all are currently selected, and that is what we want, so no changes are necessary.
7. Click the **Next** button. The *Data Import Wizard–Step 4 of 5* dialog box is displayed, as shown in Figure 10B-68. This step allows us to define column characteristics for the POSTCARDS_TEMP table. Note that in the figure, we are setting the column POSTCARDID (which will become the primary key of the new table) to **INTEGER** and **NOT NULL**.
8. We can use most of the default settings provided by the Data Import Wizard–POSTCARDS_TEMP is intended to be a temporary table used to store data before we move it to a final location, so these settings generally do not matter. For example, Figure 10B-69 shows the ARTISTNAME default column characteristics, where the

FIGURE 10B-67
The Data Import Wizard—Step 3 of 5 Dialog Box



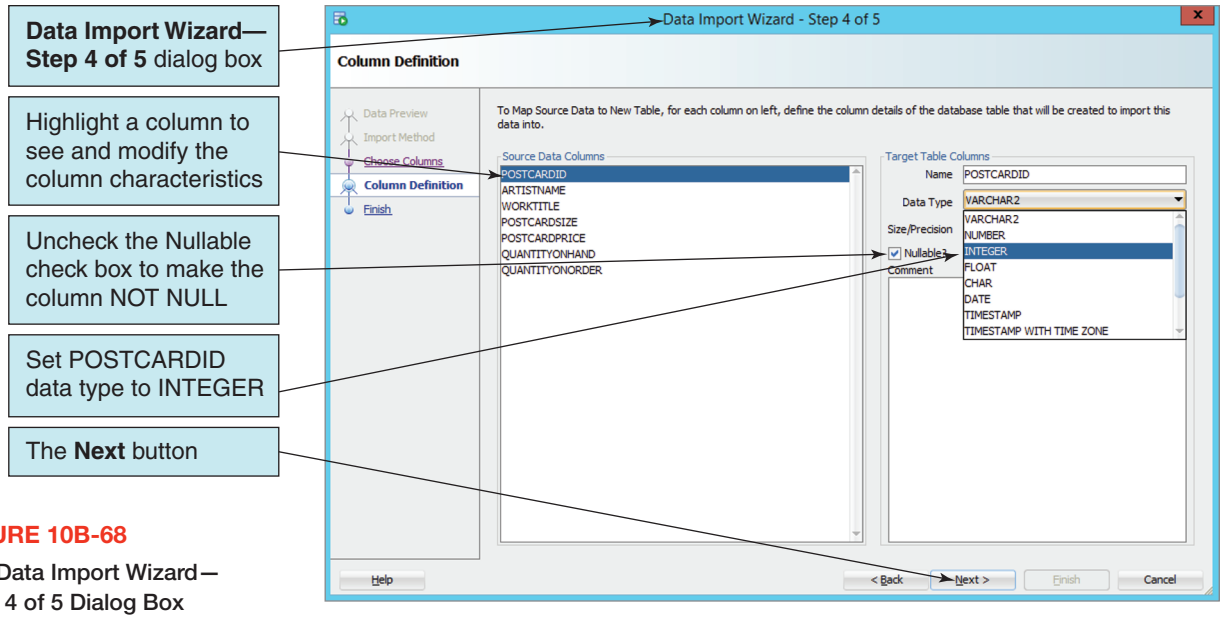


FIGURE 10B-68
The Data Import Wizard—Step 4 of 5 Dialog Box

column in POSTCARDS_TEMP will be VARCHAR2 (31) and NULL. We will accept this at this point and set the desired column characteristic in the VRG database later.

9. The only other columns that need to be edited are QUANTITYONHAND and QUANTITYONORDER, which both need to be set to **INTEGER** (but stay nullable).
10. When you have completed editing the column characteristics, click the **Next** button to display the *Data Import Wizard–Step 5 of 5* dialog box, as shown in Figure 10B-70. In this dialog box, click the **Verify** button to test the import parameters. As shown in the figure, all text statuses should be SUCCESS before the import is actually run.
11. Click the **Finish** button. The Import Data dialog box is displayed to show that the import is complete, as shown in Figure 10B-71.
12. Click the Import Data dialog box **OK** button to close the dialog box and end the import process.
13. Right-click the **Tables (Filtered)** WPC database object, and click the **Apply Filter…** command. In the Filter dialog box, add in the POSTCARDS_TEMP table

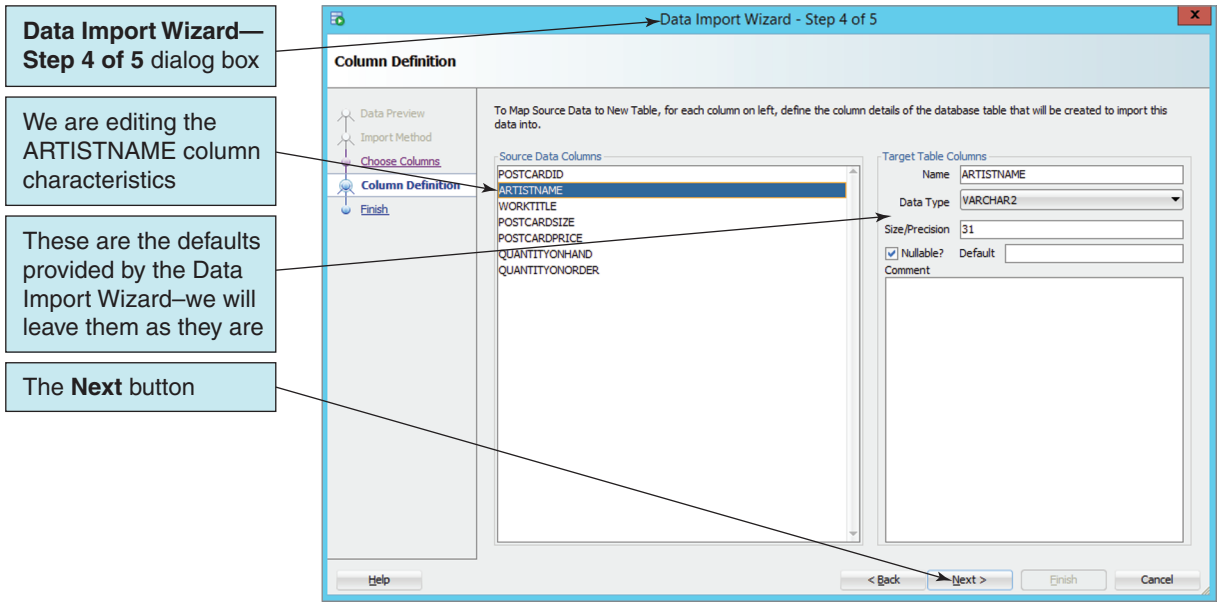


FIGURE 10B-69
The Column Definition for ARTISTNAME

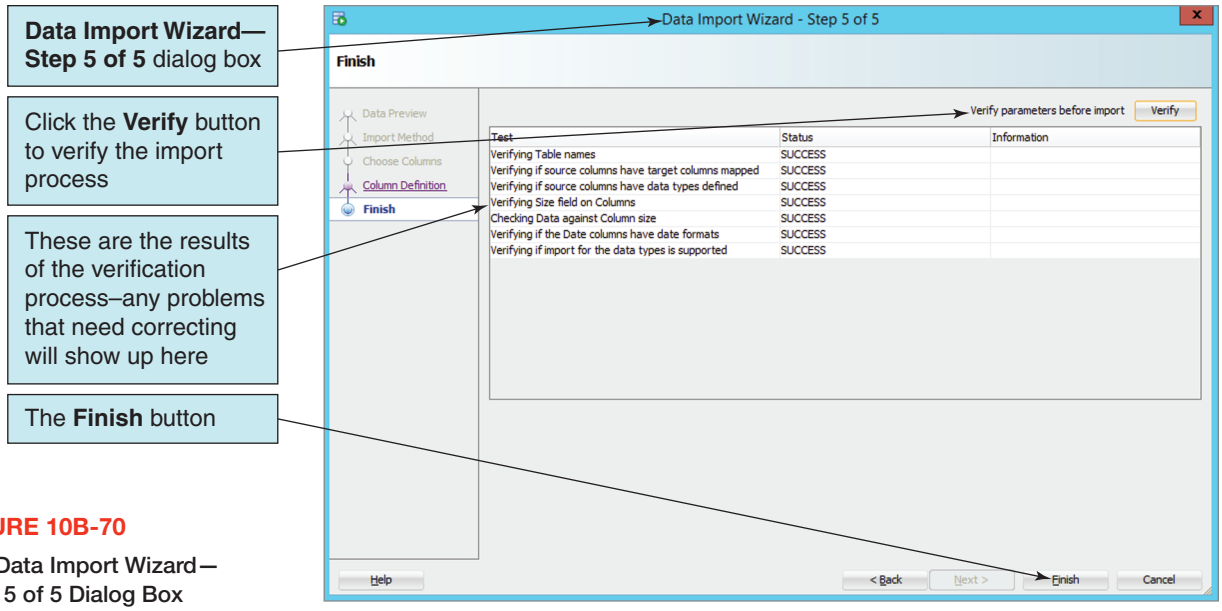


FIGURE 10B-70

The Data Import Wizard—Step 5 of 5 Dialog Box

by NAME and equals (=) to add the POSTCARDS_TEMP table to the list of visible database tables, as shown in Figure 10B-72.

14. Click the **OK** button on the Filter dialog box. The POSTCARDS_TEMP table now appears in the Tables (filtered) objects, as shown in Figure 10B-73 (where the POSTCARDS_TEMP table object itself has been expanded to show the columns).
15. In the VRG-Database SQL worksheet window, run SQL-Query-CH10B-04:

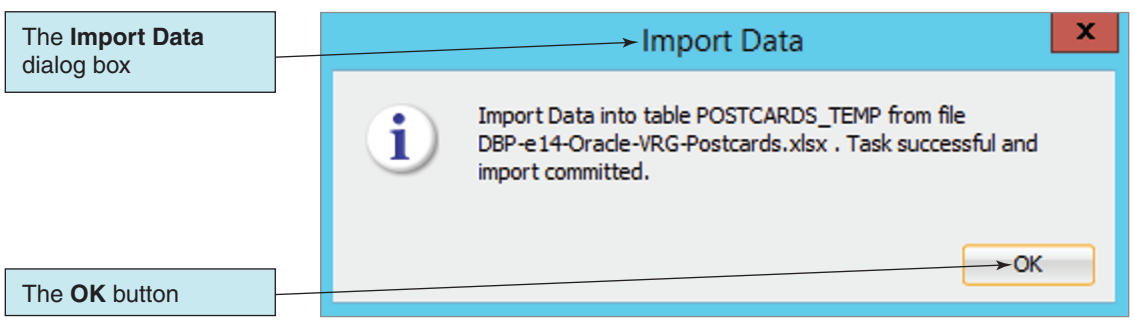
```
/* *** SQL-Query-CH10B-04 *** */
SELECT    *
FROM      COMPUTER;
```

16. The results of SQL-Query-CH10B-04 are also shown in Figure 10B-73. Note that all the columns and data are correct but that for some unknown reason the POSTCARDID primary key column is displayed as the last column in the results instead of the first.
17. The POSTCARDS_TEMP table has now been added to the WPC database.

Now that we have successfully imported the temporary *POSTCARDS_TEMP* table, we will need to design and implement the actual final table or tables in the VRG database to store this data in the form we want and then populate those tables. We will deal with these steps in Review Question 10B.40.

FIGURE 10B-71

The Import Data Dialog Box



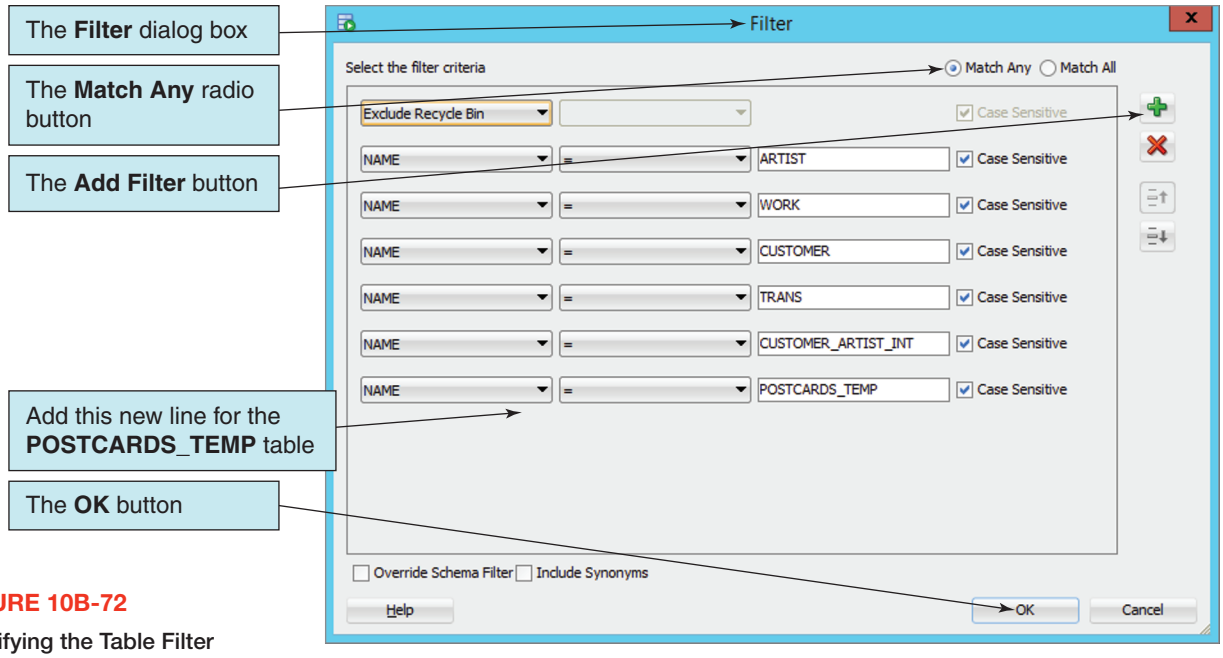


FIGURE 10B-72
Modifying the Table Filter

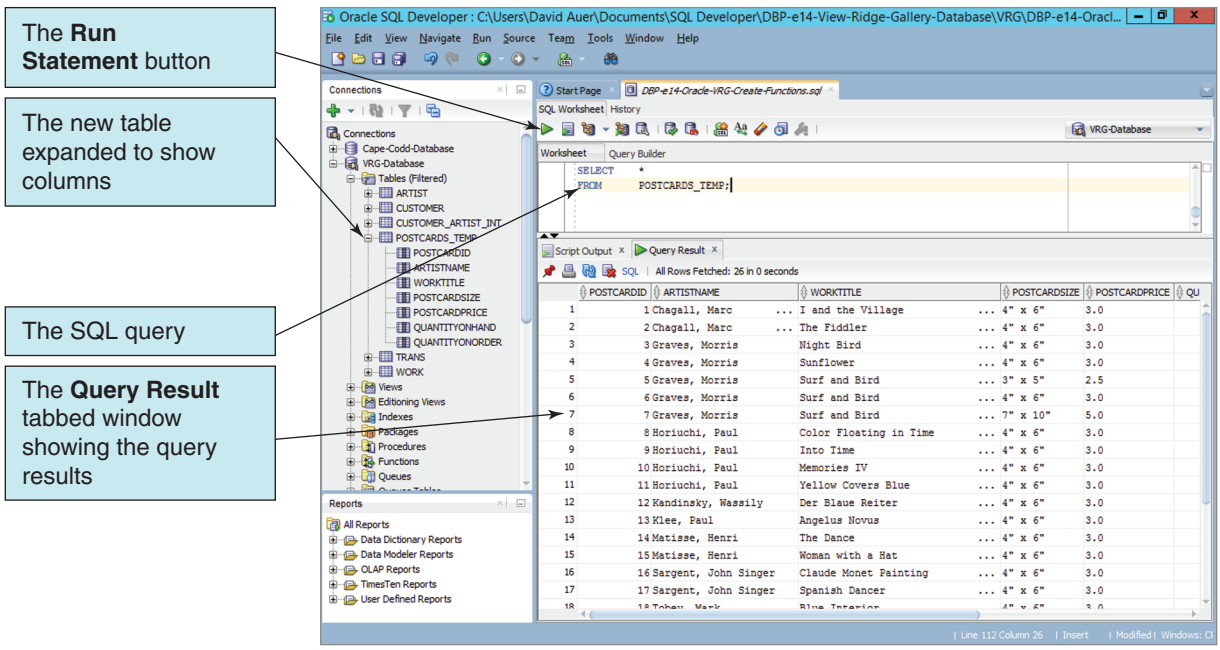
Oracle Database Application Logic

You can process an Oracle Database database from an application in a number of different ways. One is to create application code using a language such as Java, C#, C++, Visual Basic, or some other programming language and then invoke Oracle DBMS commands from those programs. The modern way to do that is to use a library of object classes, create objects that accomplish database work, and process those objects by setting object properties and invoking object methods.

Of course, we can save groups of database commands in .sql script files. Such files are then processed in SQL*Plus or SQL Developer, just as we did in creating the View Ridge database in the prior section. For security, such files should be used only during application development and testing, never on an operational database.

Another way of processing an Oracle Database database is to create stored procedures, as described in Chapter 7. These stored procedures can then be invoked from application

FIGURE 10B-73
The POSTCARDS_TEMP Table



programs or from Web pages using languages such as VBScript or JScript. Stored procedures can also be executed from SQL*Plus or SQL Developer. This should be done only when the procedures are being developed and tested, however. As described in Chapter 7, for security reasons, no one other than authorized members of the database administration staff should be allowed to interactively process an operational database.

Finally, application logic can be embedded in triggers. As you learned in Chapter 7, triggers can be used for validity checking, to set default values, to update views, and to implement referential integrity constraints.

In this chapter, we will describe and illustrate two stored procedures. We will test those procedures by invoking them from SQL Developer. Again, this should be done only during development and testing. You will learn how to invoke those stored procedures from application code in Chapter 11. We will describe four triggers, one for each of the four trigger uses. These triggers are invoked by Oracle when the specified actions occur.

Oracle Database PL/SQL

PL/SQL is Oracle Database's variant of SQL and includes Oracle Database's variant of SQL/PSM as discussed in Chapter 7, which provides the necessary procedural language extensions needed for use in stored procedures and triggers. There are certain elements of PL/SQL that we will use in such code, and therefore we need to discuss them at this point. Information on these and other PL/SQL language components can be found in the Oracle Database 12*c* Online Documentation at *https://docs.oracle.com/database/121/nav/portal_5.htm*.

PL/SQL variables and parameters are not identified by any special symbols. Thus, WorkID can be both a column name and an Oracle Database variable or parameter. It is good coding to avoid such confusions by adopting naming conventions for parameters and variables. We will use the prefix *var* at the start of our variable names and the prefix *new* for input parameters.

A **parameter** is a value that is passed to the stored procedure when it is called. A **variable** is a value used within the stored procedure itself. Comments in PL/SQL are enclosed in /* (slash asterisk) and */ (asterisk slash) signs, or follow -- (two dashes) if they are restricted to one line.

Variable values may be assigned using a **PL/SQL assignment statement** using the **PL/SQL assignment operator :=** (a colon followed immediately by an equals sign) or a **PL/SQL SELECT INTO statement**. For example, we could write the following code:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10B-01 *** */
-- Using the PL/SQL assignment statement
varNationality := 'French';
-- Using the PL/SQL SELECT INTO statement
SELECT          Nationality INTO varNationality
FROM            ARTIST
WHERE           LastName = 'Chagall';
```

PL/SQL Block Structure

PL/SQL uses a basic block structure to organize procedure and trigger code. The basic block structure is:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10B-02 *** */
DECLARE
    --{Declare variables here - this section is optional)
BEGIN
    --{Executable statements here - this section is required}
[EXCEPTION - {Optional error (exception) handling here}]
END;
/
```

The **BEGIN…END block** construction is required to define a block of PL/SQL statements in a stored procedure, function, or trigger. We can also use BEGIN…END blocks within another BEGIN…END block if doing so helps us control our code or clarify our logic.

Note that the semicolon (;) ends that block and that the slash (/) instructs Oracle Database to compile and execute the block. The slash is unique to Oracle Database and must be included in Oracle Database blocks of code.

PL/SQL Control-of-Flow Statements

Control-of-flow statements contain procedural language components that let you control exactly which parts of your code are used and the conditions required for their use. These components include IF…THEN…ELSE…END IF, WHILE, RETURN, and other keywords that can be used to direct the operations of a block of code. The **IF…THEN…ELSE…END IF keywords** are used to test for a condition and then direct which blocks of code are to be executed based on the result of that test. Note that the END IF keyword *is* used as part of this construct in PL/SQL, and this is common in many programming languages.

The **FOR keyword**, the **WHILE keyword**, and the **LOOP keyword** are used to create loops in PL/SQL, where a section of code is repeated as long as some condition is true. The **EXIT WHEN keywords** are used to exit a block of code, and the **RETURN keyword** is used to exit a block of code and terminate whatever code structure (stored procedure or trigger) is running.

As an example, let's consider a new customer at the View Ridge Gallery who needs to have customer data entered into the CUSTOMER table and artist interest data entered into the CUSTOMER_ARTIST_INT table. The new customer is Michael Bench, with phone number 206-876-8822, email address Michael.Bench@somewhere.com, and an interest in French artists. Michael will also need a CustomerID assigned to him, and we will use the seqCID sequence to generate the needed surrogate key value.

Before we enter Michael's data, we need to check to see whether Michael is already in the database. To do this, we can use the following SQL code in a stored procedure:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10B-03 *** */
BEGIN
DECLARE      varRowCount;
-- Check to see if Customer already exists in database
SELECT       COUNT(*) INTO varRowCount
FROM         CUSTOMER
WHERE        LastName = 'Bench'
    AND      FirstName = 'Michael'
    AND      EmailAddress = 'Michael.Bench@somewhere.com'
    AND      AreaCode = '206'
    AND      PhoneNumber = '876-8822';
-- IF varRowCount > 0 THEN Customer already exists.
IF (varRowCount > 0)
       THEN RETURN;
       END IF;
INSERT INTO CUSTOMER (CustomerID, LastName, FirstName, EmailAddress,
AreaCode, PhoneNumber)
     VALUES(seqCID.NextVal, 'Bench', 'Michael',
          'Michaeal.Bench@somewhere.com', '206', '876-8822');
END;
/
```

This block of SQL code illustrates the use of most of the control-of-flow keywords we've discussed except those used for looping. The WHILE and REPEAT keywords are used in code loops, and one use of a code loop is in an SQL cursor.

PL/SQL Cursor Statements

As we discussed in Chapter 7, a cursor is used so SQL results stored in a table can be processed one row at a time. Related cursor keywords include DECLARE, OPEN, FETCH, and CLOSE. The **DECLARE CURSOR keywords** are used to create a cursor, whereas the **OPEN keyword** actually starts the use of the cursor. The **FETCH keyword** is used to retrieve row data, and the **CLOSE keyword** is used to exit the use of a cursor. When using a cursor, the LOOP and WHILE keywords are used to control how long the cursor is active.

Let's consider Michael Bench's interest in French artists. The ARTIST table currently has two French artists: Henri Matisse and Marc Chagall. Therefore, we need to add new rows to CUSTOMER_ARTIST_INT, both of which will contain Michael's CustomerID number (now that he has one) and the ArtistID for each of these artists. To do this, we can use the following SQL code in a stored procedure:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10B-04 *** */
-- Add to DECLARE section:
varArtistID       Int;
varCustomerID     Int;
CURSOR            ArtistCursor   IS
                        SELECT      ArtistID
                        FROM        ARTIST
                        WHERE       Nationality = 'French';
-- GET the CustomerID surrogate key value.
-- Add this block of code after the INSERT INTO CUSTOMER statement.
SET varCustomerID = seqCID.Curval;
-- Create intersection record for each appropriate Artist.
OPEN ArtistCursor;
LOOP
     FETCH ArtistCursor INTO varArtistID;
     EXIT WHEN ArtistCursor %NOTFOUND;
          INSERT INTO CUSTOMER_ARTIST_INT (ArtistID, CustomerID)
               VALUES(varArtistID, varCustomerID);
END LOOP;
CLOSE ArtistCursor;
```

SQL-Code-Example-CH10B-04 above would be combined with the SQL-Code-Example-CH10B-03 example code on page 10B-71. As noted in the SQL-Code-Example-CH10B-04 code, the cursor declaration would be added to the DECLARE section, and the rest of the code would be added between the INSERT INTO CUSTOMER statement and the END; statement. The new code shows the steps necessary to use a cursor to add the new customer's interest in French artists to the CUSTOMER_ARTIST_INT table.

In SQL-Code-Example-CH10B-04 above, the ArtistCursor loops through the set of ArtistID values for French artists as long as there are more rows in the cursor. When the cursor has covered all the appropriate data, an SQL error occurs ("%NOTFOUND" in the code) that ends the loop. The cursor is then closed.

PL/SQL Output Statements

Oracle can use DBMS_OUTPUT.PUT_LINE ({text here}); to return output to SQL*Plus and SQL Developer. We will use this feature to produce output for our examples. To see such messages, you must execute the following prior to running the stored procedure:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH10B-05 *** */
SET SERVEROUTPUT ON
```

If you are not receiving output from your procedures, it is likely that you have not executed this statement.

User-Defined Functions

As we discussed in Chapter 7, a **user-defined function** (also known as a **stored function**) is a stored set of SQL statements that:

- is *called by name* from another SQL statement,
- may have *input parameters* passed to it by the calling SQL statement, and
- *returns an output value* to the SQL statement that called the function.

The logical process flow of a user-defined function is illustrated in Figure 7-20. SQL/PSM user-defined functions are very similar to the SQL built-in functions (COUNT, SUM, AVG, MAX, and MIN) that we discussed and used in Chapter 2, except that, as the name implies, we create them ourselves to perform specific tasks we need to do.

In Chapter 7, we used as our example the common problem of needing a name in the format *LastName, FirstName* (including the comma) in a report when the database stores the basic data in two fields named *FirstName* and *LastName*. Here we will build a user-defined function for a similar task: concatenating the *FirstName* and *LastName* fields into a name in the format *FirstName LastName* (including the space!). This construct is commonly used, for example, in mailing labels.

Using the data in the VRG database, we could, of course, simply include the code to do this in an SQL statement (similar to SQL-Query-CH02-37-Oracle-Database in Chapter 2) such as:

```
/* *** SQL-Query-CH10B-05 *** */
SELECT      RTRIM(FirstName) ||' '|| RTRIM(LastName) AS CustomerName,
            Street, City, State, ZIPorPostalCode
FROM        CUSTOMER
ORDER BY    CustomerName;
```

This produces the desired results, but at the expense of working out some cumbersome coding:

	CUSTOMERNAME	STREET	CITY	STATE	ZIPORPOSTALCODE
1	Chris Wilkens	87 Highland Drive	Olympia	WA	98508
2	David Smith	813 Tumbleweed Lane	Loveland	CO	81201
3	Donald Gray	55 Bodega Ave	Bodega Bay	CA	94923
4	Fred Smathers	10899 88th Ave	Bainbridge Island	WA	98110
5	Jeffrey Janes	123 W. Elm St	Renton	WA	98055
6	Lynda Johnson	117 C Street	Washington	DC	20003
7	Mary Beth Frederickson	25 South Lafayette	Denver	CO	80201
8	Selma Warning	205 Burnaby	Vancouver	BC	V6Z 1W2
9	Susan Wu	105 Locust Ave	Atlanta	GA	30322
10	Tiffany Twilight	88 1st Avenue	Langley	WA	98260

The alternative is to create a user-defined function to store this code. Not only does this make it easier to use, but it also makes it available for use in other SQL statements. Figure 10B-74 shows

```
CREATE FUNCTION FirstNameFirst
      -- These are the input parameters
         (
          varFirstName     Char,
          varLastName      Char
         )
      -- This is the variable that will hold the returned value
      RETURN           Varchar
      is varFullName   Varchar(60);

BEGIN

      -- SQL statements to concatenate the names in the proper order
      varFullName := (RTRIM(varFirstName) ||' '|| RTRIM(varLastName));
      -- Return the concatenated name
      RETURN varFullName;

END;
/
```

FIGURE 10B-74

The SQL Statements for the FirstNameFirst User-Defined Function

a user-defined function named *FirstNameFirst* written in PL/SQL for use with Oracle Database, and the SQL code for the function uses, as we would expect, specific syntax requirements for Oracle Database PL/SQL:

- The function is created and stored in the database by using the **PL/SQL CREATE FUNCTION statement**.
- The variable names of both the input parameters and the returned output value start with *var* to indicate that these are variable names.
- The concatenation syntax is PL/SQL syntax.

Now that we have created and stored the user-defined function, we can use it in SQL-Query-CH10B-06:

```
/* *** SQL-Query-CH10B-06 *** */
SELECT      FirstNameFirst(FirstName, LastName) AS CustomerName,
            Street, City, State, ZIPorPostalCode
FROM        CUSTOMER
ORDER BY    CustomerName;
```

Now we have a function that produces the results we want, which of course are identical to the results for SQL-QUERY-CH10B-05 above:

	CUSTOMERNAME	STREET	CITY	STATE	ZIPORPOSTALCODE
1	Chris Wilkens	87 Highland Drive	Olympia	WA	98508
2	David Smith	813 Tumbleweed Lane	Loveland	CO	81201
3	Donald Gray	55 Bodega Ave	Bodega Bay	CA	94923
4	Fred Smathers	10899 88th Ave	Bainbridge Island	WA	98110
5	Jeffrey Janes	123 W. Elm St	Renton	WA	98055
6	Lynda Johnson	117 C Street	Washington	DC	20003
7	Mary Beth Frederickson	25 South Lafayette	Denver	CO	80201
8	Selma Warning	205 Burnaby	Vancouver	BC	V6Z 1W2
9	Susan Wu	105 Locust Ave	Atlanta	GA	30322
10	Tiffany Twilight	88 1st Avenue	Langley	WA	98260

The advantage of having a user-defined function is that we can now use it whenever we need to without having to re-create the code. For example, our previous query used data in

the View Ridge Gallery CUSTOMER table, but we could just as easily use the function with the data in the ARTIST table:

```
/* *** SQL-Query-CH10B-07 *** */
SELECT      FirstNameFirst(FirstName, LastName) AS ArtistName,
            DateOfBirth, DateDeceased
FROM        ARTIST
ORDER BY    ArtistName;
```

This query produces the expected results:

	ARTISTNAME	DATEOFBIRTH	DATEDECEASED
1	Henri Matisse	1869	1954
2	Joan Miro	1893	1983
3	John Singer Sargent	1856	1925
4	Marc Chagall	1887	1985
5	Mark Tobey	1890	1976
6	Morris Graves	1920	2001
7	Paul Horiuchi	1906	1999
8	Paul Klee	1879	1940
9	Wassily Kandinsky	1866	1944

We can even use the function multiple times in the same SQL statement, as shown in SQL-Query-CH10B-08, which is a variant on the SQL query we used to create the SQL view CustomerInterestsView in our discussion of SQL views in this chapter:

```
/* *** SQL-Query-CH10B-08 *** */
SELECT      FirstNameFirst(C.FirstName, C.LastName) AS CustomerName,
            FirstNameFirst(A.FirstName, A.LastName) AS ArtistName
FROM        CUSTOMER C JOIN CUSTOMER_ARTIST_INT CAI
  ON        C.CustomerID = CAI.CustomerID
            JOIN      ARTIST A
               ON     CAI.ArtistID = A.ArtistID
ORDER BY    CustomerName, ArtistName;
```

This query produces the expected large result that is shown in Figure 10B-75, where we see both CustomerName and ArtistName display the names in the FirstName LastName syntax produced by the *FirstNameFirst* user-defined function. Compare the results in this figure to those in Figure 10B-61, which presents similar results, but without the formatting provided by the *FirstNameFirst* function.

Having dealt with the problem of concatenating two separate name values into one, let's consider the opposite problem: separating a combined name into separate elements. This is a problem that commonly occurs with data provided in a Microsoft Excel worksheet, where the user simply put an entire name into one cell. As a practical example of this, consider the VRG *POSTCARDS_TEMP* table we imported in our discussion of how to import Microsoft Excel data into a table. Looking at Figure 10B-73, we can see that the ArtistName column contains the combined artist name in *LastName, FirstName* format.

Because a best practice of database design is to separate data like this into its separate elements, we have the problem of breaking this data into *LastName* and *FirstName*. We can use a user-defined function to do this.

Note that the delineator or separator between LastName and FirstName is a comma. We can search for the comma using the Oracle Database built-in character

FIGURE 10B-75

Results of SQL Query Using the FirstNameFirst User-Defined Function

	CUSTOMERNAME	ARTISTNAME
1	Chris Wilkens	Mark Tobey
2	Chris Wilkens	Morris Graves
3	Chris Wilkens	Paul Horiuchi
4	David Smith	Henri Matisse
5	David Smith	Joan Miro
6	David Smith	John Singer Sargent
7	David Smith	Marc Chagall
8	David Smith	Wassily Kandinsky
9	Donald Gray	Mark Tobey
10	Donald Gray	Morris Graves
11	Donald Gray	Paul Horiuchi
12	Fred Smathers	Mark Tobey
13	Fred Smathers	Morris Graves
14	Fred Smathers	Paul Horiuchi
15	Jeffrey Janes	Mark Tobey
16	Jeffrey Janes	Morris Graves
17	Jeffrey Janes	Paul Horiuchi
18	Mary Beth Frederickson	Henri Matisse
19	Mary Beth Frederickson	Joan Miro
20	Mary Beth Frederickson	Marc Chagall
21	Mary Beth Frederickson	Wassily Kandinsky
22	Selma Warning	John Singer Sargent
23	Selma Warning	Marc Chagall
24	Selma Warning	Morris Graves
25	Tiffany Twilight	John Singer Sargent
26	Tiffany Twilight	Mark Tobey
27	Tiffany Twilight	Morris Graves
28	Tiffany Twilight	Paul Horiuchi

string function **INSTR**, which will return the numeric position of the comma. The full syntax of the function is:

```
INSTR (ExpressionToSearch, ExpressionToFind [, StartLocation])
```

In the *POSTCARDS_TEMP* table, we want to find the comma (",") in ArtistName starting at the default location of 1 (character strings are counted starting at 1, not 0).

Once we have found the comma, we can retrieve the last name by using the SQL Server built-in character string function **SUBSTR**, which will return a subset of the characters from a character string. The full syntax of the function is:

```
SUBSTR (ExpressionToSearch, StartLocation, Length)
```

In the *POSTCARDS_TEMP* table, we want to return the subset of ArtistName starting at 1 and ending one character below the comma ([Value returned by INSTR] - 1). We put these together into a user-defined function named *GetLastNameCommaSeparated* as shown in Figure 10B-76.

FIGURE 10B-76
The SQL Statements for the GetLastNameCommaSeparated User-Defined Function

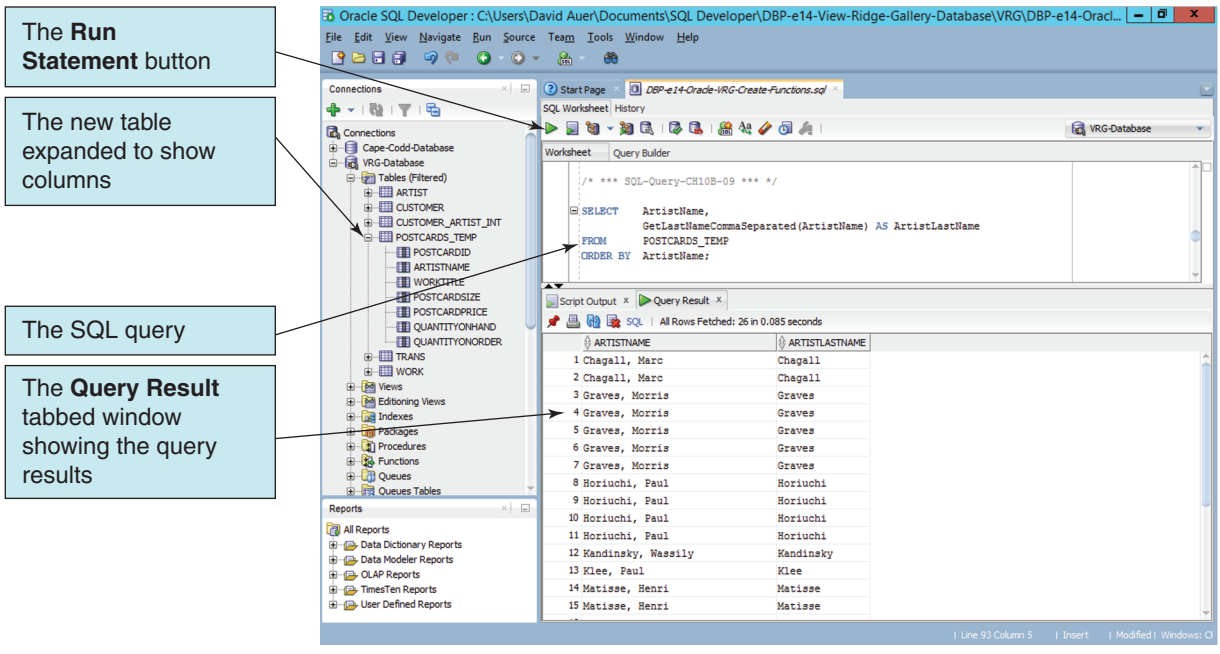
```
CREATE OR REPLACE FUNCTION GetLastNameCommaSeparated
   -- These are the input parameters
   (
      varName         Varchar
   )

   -- This is the variable that will hold the returned value
   RETURN          Varchar
   IS
   -- This is the variable that will hold the value to be returned
   varLastName     Varchar(25);
   -- This is the variable that will hold the position of the comma
   varIndexValue   INT;

BEGIN

   -- SQL statement to find the comma separator
   varIndexValue := INSTR(varName, ',');

   -- SQL statement to determine last name
   varLastName := SUBSTR(varName, 1, (varIndexValue - 1));

   -- Return the last name
   RETURN varLastName;

END;
/
```

Now that we have created and stored the user-defined function, we can use it in SQL-Query-CH10B-09:

```
/* *** SQL-Query-CH10B-09 *** */
SELECT      ArtistName,
            GetLastNameCommaSeparated(ArtistName) AS ArtistLastName
FROM        POSTCARDS_TEMP
ORDER BY    ArtistName;
```

FIGURE 10B-77
The Results for SQL-Query-CH10B-09

The results shown in Figure 10B-77 are exactly what we want, with the ArtistLastName column containing only the last name.



Now that we can determine the last name of the artists in the POSTCARDS_TEMP table, let's return to our discussion of that table and how we will integrate the data in it into the VRG database. Because the ARTIST table uses ArtistID as the primary key and WORK uses WorkID for the primary key, we have to find some way of associating these primary key values with the data in POSTCARDS_TEMP. We can use the *GetLastNameCommaSeparated* user-defined function to help us do this.

First, we need to alter the *POSTCARDS_TEMP* table by adding a column named *ArtistLastName* to hold the last name values. The full discussion of how to do this is in Chapter 8 on database redesign, where we discuss how to use the **SQL ALTER TABLE statement**. Here we will use it to add an ArtistLastName column and an ArtistID column.

```
/* *** SQL-ALTER-TABLE-CH10B-01 *** */
ALTER TABLE POSTCARDS_TEMP
   ADD      ArtistLastName     Char(25)      NULL;
ALTER TABLE POSTCARDS_TEMP
   ADD      ArtistID           Int           NULL;
```

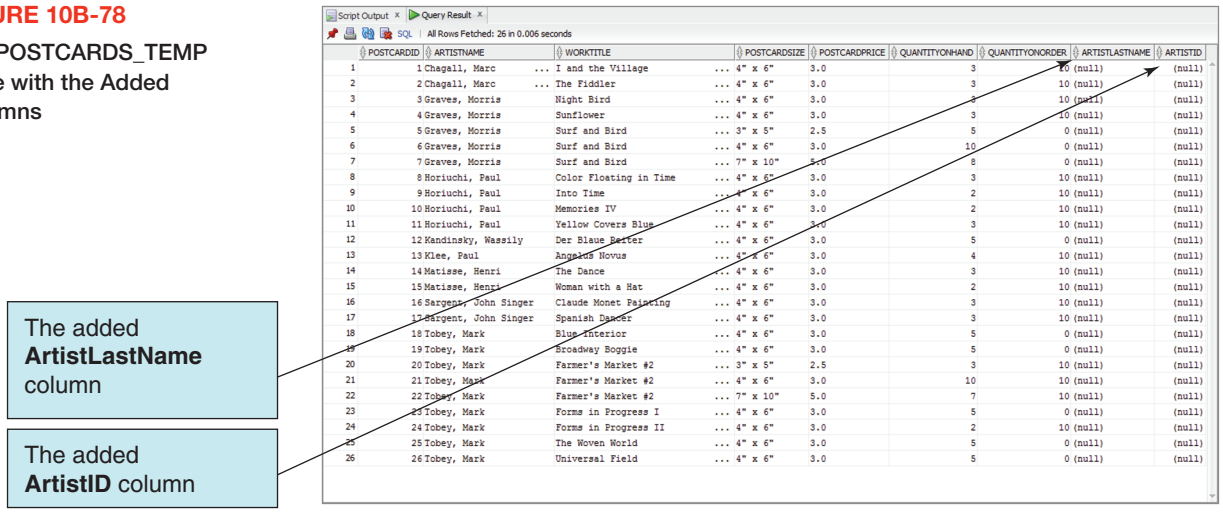
Note that we allow the values of both these columns to be NULL because we have not inserted any data and therefore we cannot create them as NOT NULL even if that is what we ultimately want them to be (see Chapter 8 for a discussion of the steps to add a NOT NULL column to a table). The POSTCARDS_TEMP table now appears as shown in Figure 10B-78, with the two new columns displayed at the right side of the table:

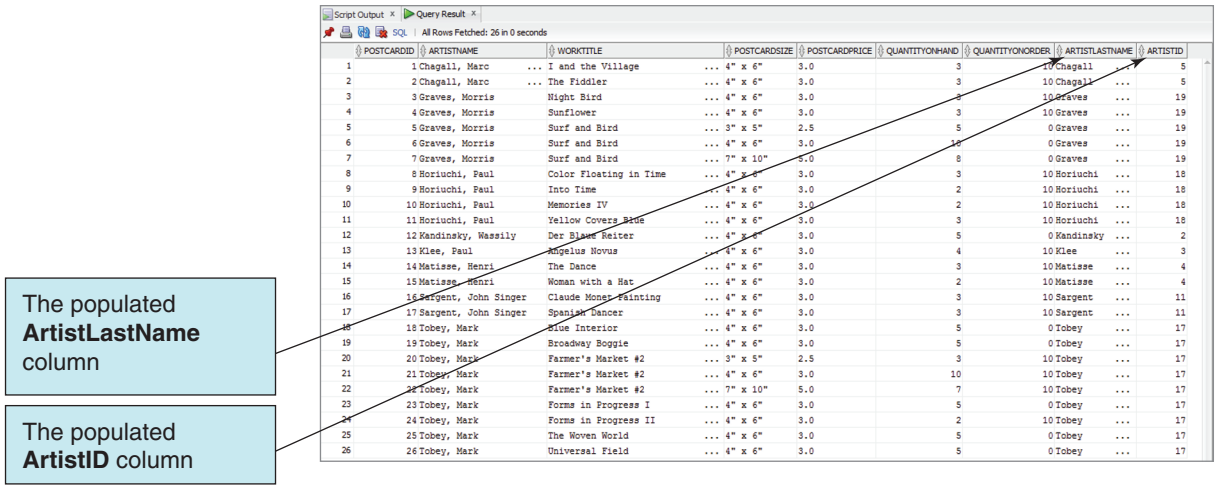
Now we are ready populate the POSTCARDS_TEMP table with artist last names and artist IDs. The SQL statements to do this are:

```
/* *** SQL-UPDATE-CH10B-01 *** */
UPDATE      POSTCARDS_TEMP
   SET      ArtistLastName = GetLastNameCommaSeparated(ArtistName);
/* *** SQL-UPDATE-CH10B-02 *** */
UPDATE      POSTCARDS_TEMP
   SET      ArtistID =
            (SELECT  ArtistID
             FROM    ARTIST
             WHERE   ARTIST.LastName = POSTCARDS_TEMP.ArtistLastName);
```

FIGURE 10B-78

The POSTCARDS_TEMP Table with the Added Columns





	POSTCARDID	ARTISTNAME	WORKTITLE	POSTCARDSIZE	POSTCARDPRICE	QUANTITYONHAND	QUANTITYONORDER	ARTISTLASTNAME	ARTISTID
1	1	Chagall, Marc	I and the Village	4" x 6"	3.0	3	10	Chagall	5
2	2	Chagall, Marc	The Fiddler	4" x 6"	3.0	3	10	Chagall	5
3	3	Graves, Morris	Night Bird	4" x 6"	3.0	3	10	Graves	19
4	4	Graves, Morris	Sunflower	4" x 6"	3.0	3	10	Graves	19
5	5	Graves, Morris	Surf and Bird	3" x 5"	2.5	5	0	Graves	19
6	6	Graves, Morris	Surf and Bird	4" x 6"	3.0	10	0	Graves	19
7	7	Graves, Morris	Surf and Bird	7" x 10"	5.0	8	0	Graves	19
8	8	Horiuchi, Paul	Color Floating in Time	4" x 6"	3.0	3	10	Horiuchi	18
9	9	Horiuchi, Paul	Into Time	4" x 6"	3.0	2	10	Horiuchi	18
10	10	Horiuchi, Paul	Memories IV	4" x 6"	3.0	2	10	Horiuchi	18
11	11	Horiuchi, Paul	Yellow Covers Blue	4" x 6"	3.0	3	10	Horiuchi	18
12	12	Kandinsky, Wassily	Der Blaue Reiter	4" x 6"	3.0	5	0	Kandinsky	2
13	13	Klee, Paul	Angelus Novus	4" x 6"	3.0	4	10	Klee	3
14	14	Matisse, Henri	The Dance	4" x 6"	3.0	3	10	Matisse	4
15	15	Matisse, Henri	Woman with a Hat	4" x 6"	3.0	2	10	Matisse	4
16	16	Sargent, John Singer	Claude Monet Painting	4" x 6"	3.0	3	10	Sargent	11
17	17	Sargent, John Singer	Spanish Dancer	4" x 6"	3.0	3	10	Sargent	11
18	18	Tobey, Mark	Blue Interior	4" x 6"	3.0	5	0	Tobey	17
19	19	Tobey, Mark	Broadway Boggie	4" x 6"	3.0	5	0	Tobey	17
20	20	Tobey, Mark	Farmer's Market #2	3" x 5"	2.5	3	10	Tobey	17
21	21	Tobey, Mark	Farmer's Market #2	4" x 6"	3.0	10	10	Tobey	17
22	22	Tobey, Mark	Farmer's Market #2	7" x 10"	5.0	7	10	Tobey	17
23	23	Tobey, Mark	Forms in Progress I	4" x 6"	3.0	5	0	Tobey	17
24	24	Tobey, Mark	Forms in Progress II	4" x 6"	3.0	2	10	Tobey	17
25	25	Tobey, Mark	The Woven World	4" x 6"	3.0	5	0	Tobey	17
26	26	Tobey, Mark	Universal Field	4" x 6"	3.0	5	0	Tobey	17

FIGURE 10B-79

The POSTCARDS_TEMP Table with the Populated Columns

Note the use of the *GetLastNameCommaSeparated* user-defined function in the SQL-UPDATE-CH10B-01. Also note the use of an SQL subquery in SQL-UPDATE-CH10B-02, which illustrates that SQL subqueries can be used in SQL statements beyond just the SQL SELECT statement. In fact, we can use an SQL subquery in INSERT, UPDATE, DELETE, and MERGE statements, and it often is exactly what we need! Finally, note that SQL-UPDATE-CH10B-02 only works if all the artists have unique last names. The updated *POSTCARDS_TEMP* table is shown in Figure 10B-79, with the new column data displayed in the new columns. We still have work to do to integrate the *POSTCARDS_TEMP* table data into the VRG database, and we will continue that work in Review Question 10B.70.

Stored Procedures

As with other database structures, you can write a stored procedure in an SQL script text file and process the commands using the SQL Developer. The first time you create a stored procedure in an SQL script, you use the **SQL CREATE PROCEDURE statement**. In Oracle, however, always use the phrase **CREATE OR REPLACE** in this statement when developing PL/SQL procedures, as shown in Figure 10B-85, to allow for changes to the procedure. Use the **SQL DROP PROCEDURE statement** to delete the procedure.

A stored procedure is a PL/SQL or Java program that is stored within the database. Stored procedures are programs; they can have parameters, they can invoke other procedures and functions, they can return values, and they can raise exceptions. Stored procedures can be invoked remotely. Here we will consider two stored procedure examples.

The Stored Procedure InsertCustomerAndInterests

In our previous discussion of PL/SQL, we used as our example the need to enter data for a new customer and the artists of interest to that customer. The code segments we wrote were very specifically tied to the data we used and thus of limited use. Is there a way to write a general block of code that could be used for more than one customer? Yes, and that block of code is a stored procedure.

Figure 10B-80 shows the PL/SQL code for the InsertCustomerAndInterests stored procedure. This stored procedure generalizes our previous code and can be used to insert data for any new customer into CUSTOMER and then store data for that customer in CUSTOMER_ARTIST_INT, linking the customer to all artists having a particular nationality.

Six parameters are input to the procedure: newLastName, newFirstName, newEmailAddress, newAreaCode, newPhoneNumber, and newNationality. The first five parameters are the new customer data, and the sixth one is the nationality of the artists that the new customer has an interest in. The stored procedure also used three variables: varRowCount, varArtistID, and varCustomerID. These variables are used to store the

```
CREATE OR REPLACE
PROCEDURE InsertCustomerAndInterests
          (
           newLastName         IN Char,
           newFirstName        IN Char,
           newEmailAddress     IN VarChar,
           newAreaCode         IN Char,
           newPhoneNumber      IN Char,
           newNationality      IN Char
          )
AS


       varRowCount          Int;
       varArtistID          Int;
       varCustomerID        Int;
       CURSOR ArtistCursor IS
              SELECT    ArtistID
              FROM      ARTIST
              WHERE     Nationality=newNationality;


BEGIN

   -- Check to see if Customer already exists in database

   SELECT    COUNT(*) INTO varRowCount
   FROM      CUSTOMER
   WHERE     LastName = newLastName
      AND    FirstName = newFirstName
      AND    EmailAddress = newEmailAddress
      AND    AreaCode = newAreaCode
      AND    PhoneNumber = newPhoneNumber;


   -- IF varRowCount > 0 THEN Customer already exists.
   IF (varRowCount > 0) THEN
      BEGIN
         DBMS_OUTPUT.PUT_LINE('************************************************');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   The Customer is already in the database. ');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   Customer Last Name         =  '||newLastName);
         DBMS_OUTPUT.PUT_LINE('   Customer First Name        =  '||newFirstName);
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('************************************************');
         RETURN;
      END;
   END IF;
   -- IF @RowCount = 0 THEN Customer does not exist in database.
   IF (varRowCount = 0) THEN
      BEGIN
      -- Insert new Customer data.
         INSERT INTO CUSTOMER
             (CustomerID, LastName, FirstName, EmailAddress, AreaCode, PhoneNumber)
             VALUES(seqCID.NextVal, newLastName, newFirstName, newEmailAddress,
                    newAreaCode, newPhoneNumber);
         -- Get new CustomerID surrogate key value.
         varCustomerID := seqCID.CurrVal;
```

FIGURE 10B-80

The SQL Statements for the InsertCustomerAndInterests Stored Procedure

```
            DBMS_OUTPUT.PUT_LINE('************************************************');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   The new Customer is now in the database. ');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   Customer Last Name         =  '||newLastName);
            DBMS_OUTPUT.PUT_LINE('   Customer First Name        =  '||newFirstName);
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('************************************************');


            -- Create intersection record for each appropriate Artist.
            --Process each appropriate Artist
            OPEN ArtistCursor;
               LOOP
                  FETCH ArtistCursor INTO varArtistID;
                  EXIT WHEN ArtistCursor%NOTFOUND;
                  INSERT INTO CUSTOMER_ARTIST_INT
                     (ArtistID, CustomerID)
                     VALUES(varArtistID, varCustomerID);
            DBMS_OUTPUT.PUT_LINE('************************************************');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   New CUSTOMER_ARTIST_INT row added. ');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   ArtistID        =  '||varArtistID);
            DBMS_OUTPUT.PUT_LINE('   CustomerID      =  '||varCustomerID);
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('************************************************');
               END LOOP;
            CLOSE ArtistCursor;
         END;
      END IF;
      COMMIT;
   END;
   /
```

FIGURE 10B-80
Continued

number of rows in a table, the value of the ArtistID primary key, and the value of the CustomerID primary key, respectively.

The first task performed by this stored procedure is to determine whether the customer already exists. If the value of varRowCount in the first SELECT statement is greater than zero, a row for that customer already exists. In this case, nothing is done, and the stored procedure prints an error message and exits (using the RETURN command). As noted earlier, the error message is visible in SQL Developer, but it generally would not be visible to application programs that invoked this procedure. Instead, a parameter or other facility needs to be used to return the error message back to the user via the application program. That topic is beyond the scope of the present discussion, but we will send a message back to SQL Developer to mimic such actions and provide a means to make sure our stored procedures are working correctly.

If the customer does not already exist, the procedure inserts the new data into the table CUSTOMER, and then the new customer's CustomerID is read into the variable varCustomerID. To create the appropriate intersection of table rows, an SQL cursor named ArtistCursor is created on an SQL statement that obtains all ARTIST rows where Nationality equals the parameter newNationality. The cursor is opened and positioned on the first row by calling FETCH, and then the cursor is processed in a loop. In this loop, statements between BEGIN and END are iterated until Oracle signals the end of data by the value of the Oracle function %NOTFOUND. At each iteration of the loop, a new row is inserted into the intersection table CUSTOMER_ARTIST_INT.

The remainder of the procedure in Figure 10B-80 inserts the new customer data and then loops through all artists of the given nationality. This construct does several tasks. It opens the cursor and fetches the first row. Then it iterates through all of the rows in the cursor; when there are no more rows, it closes the cursor. After you execute this statement, you should query the CUSTOMER, ARTIST, and CUSTOMER_ ARTIST_INT tables to ensure that the changes were made correctly.

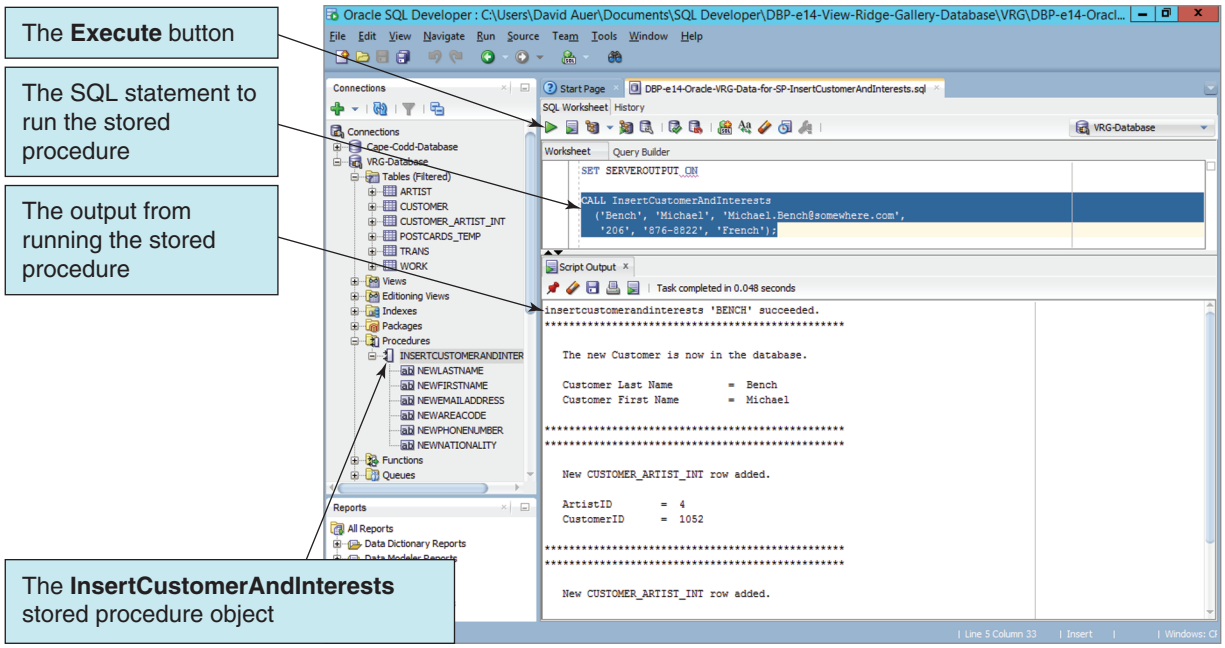


FIGURE 10B-81

Running the InsertCustomerAndInterests Stored Procedure

If you have execution-time errors, the line numbers reported differ from the line numbers you see in your text editor. You can adjust these line numbers to conform to yours, but the process is too complicated to describe here. For the simple procedures we create here, just work around the issue. Do not assume that the line numbers match, however.

To invoke the InsertCustomerAndInterests stored procedure for Michael Bench, we use the following statements:

```
SET SERVEROUTPUT ON
/* *** SQL-CALL-CH10B-01 *** */
CALL InsertCustomerAndInterests
     ('Bench', 'Michael', 'Michael.Bench@somewhere.com',
      '206', '876-8822', 'French');
```

Figure 10B-81 shows the execution of the stored procedure in SQL Developer. Notice how our sections of DBMS_OUTPUT.PUT_LINE commands have produced the necessary output so we can see what actions were taken. If we now wanted to check the tables themselves, we could do so, but that is not necessary at this point.

The Stored Procedure InsertCustomerWithTransaction

Figure 10B-82 shows a second stored procedure for recording a new customer and the sale of a work to that customer. The logic of this procedure, named InsertCustomerWithTransaction, is as follows. First, create the new customer data and then search for the TRANSACTION row for the purchased work that has a NULL value for SalesPrice. That search involves the ARTIST, WORK, and TRANSACTION tables because the Name of the artist is stored in ARTIST and the Title and Copy of the work are stored in WORK. If one, and only one, such row is found, update CustomerID, SalesPrice, and DateSold in that row. Then insert a row in the intersection table to record the customer's interest in this artist. Otherwise, make no changes to the database.

InsertCustomerWithTransaction accepts parameters having customer and purchase data as shown. Next, several variables are declared.

```
CREATE OR REPLACE
PROCEDURE InsertCustomerWithTransaction
          (
           newCustomerLastName          IN Char,
           newCustomerFirstName         IN Char,
           newCustomerEmailAddress      IN VarChar,
           newCustomerAreaCode          IN Char,
           newCustomerPhoneNumber       IN Char,
           varArtistLastName            IN Char,
           varWorkTitle                 IN Char,
           varWorkCopy                  IN Char,
           newTransSalesPrice           IN Number
          )
AS

DECLARE   varRowCount            Int;
          varAID                 Int;
          varCID                 Int;
          varWID                 Int;
          varTID                 Int;

BEGIN
   -- Start transaction - Rollback everything if unable to complete it.
   -- Check to see if Customer already exists in database

   SELECT COUNT(*) INTO varRowCount
   FROM      CUSTOMER
   WHERE     LastName = newCustomerLastName
      AND    FirstName = newCustomerFirstName
      AND    EmailAddress = newCustomerEmailAddress
      AND    AreaCode = newCustomerAreaCode
      AND    PhoneNumber = newCustomerPhoneNumber;

   -- IF varRowCount > 0 THEN Customer already exists.
   IF (varRowCount > 0) THEN
      BEGIN
         DBMS_OUTPUT.PUT_LINE('************************************************');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   The Customer is already in the database. ');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   Customer Last Name  =  '||newCustomerLastName);
         DBMS_OUTPUT.PUT_LINE('   Customer First Name =  '||newCustomerFirstName);
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('************************************************');
         RETURN;
      END;

   -- IF varRowCount = 0 THEN Customer does not exist in database.
      ELSE BEGIN
         -- Insert new Customer data.
         INSERT INTO CUSTOMER
            (CustomerID, LastName, FirstName, EmailAddress, AreaCode, PhoneNumber)
            VALUES(seqCID.NextVal, newCustomerLastName, newCustomerFirstName,
            , newCustomerEmailAddress, newCustomerAreaCode, newCustomerPhoneNumber);
         -- Get new CustomerID surrogate key value.
         varCID := seqCID.currVal;
```

(*continued*)

FIGURE 10B-82

The SQL Statements for the InsertCustomerWithTransaction Stored Procedure

```
-- Get ArtistID surrogate key value, check for validity.
SELECT    ArtistID INTO varAID
FROM      ARTIST
WHERE     LastName = varArtistLastName;

IF (varAID IS NULL) THEN
   BEGIN
DBMS_OUTPUT.PUT_LINE('****************************************');
DBMS_OUTPUT.PUT_LINE('');
DBMS_OUTPUT.PUT_LINE('   Invalid ArtistID. ');
DBMS_OUTPUT.PUT_LINE('');
DBMS_OUTPUT.PUT_LINE('****************************** *************');
       ROLLBACK;
       RETURN;
   END;
END IF;

-- Get WorkID surrogate key value, check for validity.
SELECT    WorkID INTO varWID
FROM      WORK
WHERE     ArtistID = varAID
   AND    Title = varWorkTitle
   AND    Copy = varWorkCopy;

IF varWID IS NULL THEN
   BEGIN
DBMS_OUTPUT.PUT_LINE('****************************************');
DBMS_OUTPUT.PUT_LINE('');
DBMS_OUTPUT.PUT_LINE('   Invalid WorkID. ');
DBMS_OUTPUT.PUT_LINE('');
DBMS_OUTPUT.PUT_LINE('****************************************');
       ROLLBACK;
       RETURN;
   END;
END IF;

-- Get TransactionID surrogate key value, check for validity.
SELECT    TransactionID INTO varTID
FROM      TRANS
WHERE     WorkID = varWID
   AND    SalesPrice IS NULL;

IF varTID IS NULL THEN
   BEGIN
DBMS_OUTPUT.PUT_LINE('****************************************');
DBMS_OUTPUT.PUT_LINE('');
DBMS_OUTPUT.PUT_LINE('   Invalid TransactionID.  ');
DBMS_OUTPUT.PUT_LINE('');
DBMS_OUTPUT.PUT_LINE('****************************************');
       ROLLBACK;
       RETURN;
   END;
END IF;

-- All surrogate key values of OK, complete the transaction
-- Update TRANS row
UPDATE TRANS
SET    DateSold = SysDate,
       SalesPrice = newTransSalesPrice,
       CustomerID = varCID
WHERE  TransactionID = varTID;
-- Create CUSTOMER_ARTIST_INT row
INSERT INTO CUSTOMER_ARTIST_INT (CustomerID, ArtistID)
       VALUES(varCID, varAID);
```

FIGURE 10B-82
Continued

```
        -- Commit the Transaction
        -- COMMIT;
        -- The transaction is completed. Print output
            -- Print Customer results.
            DBMS_OUTPUT.PUT_LINE('************************************************');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   The new Customer is now in the database. ');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   Customer Last Name  =  '||newCustomerLastName);
            DBMS_OUTPUT.PUT_LINE('   Customer First Name =  '||newCustomerFirstName);
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('************************************************');
            -- Print Transaction result
            DBMS_OUTPUT.PUT_LINE('************************************************');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   Transaction complete. ');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   TransactionID  =  '||varTID);
            DBMS_OUTPUT.PUT_LINE('   ArtistID       =  '||varAID);
            DBMS_OUTPUT.PUT_LINE('   WorkID         =  '||varWID);
            DBMS_OUTPUT.PUT_LINE('   Sales Price    =  '||newTransSalesPrice);
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('************************************************');
            -- Print CUSTOMER_ARTIST_INT update
            DBMS_OUTPUT.PUT_LINE('************************************************');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   New CUSTOMER_ARTIST_INT row added. ');
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('   ArtistID       =  '||varAID);
            DBMS_OUTPUT.PUT_LINE('   CustomerID     =  '||varCID);
            DBMS_OUTPUT.PUT_LINE('');
            DBMS_OUTPUT.PUT_LINE('************************************************');
        END;
    END IF;
END;
/
```

FIGURE 10B-82
Continued

The procedure first checks to see whether the input customer data already exist in the database. If not, it inserts the new customer data. In PL/SQL, there is no BEGIN TRANSACTION statement; the first database action automatically starts a transaction. Here inserting the customer data starts a new transaction.

After the customer data are inserted, the procedure finds the artist and work IDs associated with the input parameters. If there are no matching artist or work records, then the procedure will exit without committing any changes. (Note that this only works if artist last names are unique—a more thorough solution would include ArtistFirstName as a parameter to guarantee uniqueness.) Once we have the ArtistID and WorkID, we find the TransactionID; if the work has already been sold, then the procedure exits without committing any changes. If all is well up to this point, the appropriate TRANS row is updated.

Note the use of the function SysDate to store the current date. Finally, an intersection row is inserted for this customer and the artist of the purchased work (varAID).

To test this procedure, it is convenient to first define a view that shows customer purchases, such as the following:

```
/* *** SQL-CREATE-VIEW-CH10B-01 *** */
CREATE VIEW WorkPurchaseView AS
    SELECT    CUSTOMER.LastName AS CustomerLastName,
              CUSTOMER.FirstName AS CustomerFirstName,
              ARTIST.LastName as ArtistName,
              WORK.Title, WORK.Copy,
              TRANS.DateSold, TRANS.SalesPrice
```

```
    FROM      CUSTOMER JOIN TRANS
        ON    CUSTOMER.CustomerID = TRANS.CustomerID
              JOIN WORK
              ON   TRANS.WorkID = WORK.WorkID
                   JOIN   ARTIST
                          ON   WORK.ArtistID = ARTIST.ArtistID;
```

To use the InsertCustomerWithTransaction stored procedure, we will record the following purchase by our next new customer, Melinda Gliddens, who just bought a print of John Singer Sargent's *Spanish Dancer* for $350.00. The SQL statement is:

```
SET SERVEROUTPUT ON
/* *** SQL-CALL-CH10B-02 *** */
CALL InsertCustomerWithTransaction
  ('Gliddens', 'Melinda', 'Melinda.Gliddens@somewhere.com',
   '360', '765-8877',
   'Sargent', 'Spanish Dancer', '588/750', 350.00);
```

Figure 10B-83 shows the invocation of this procedure using sample data, and we can now use the WorkPurchaseView to see the new CUSTOMER and WORK in the database:

```
/* *** SQL-Query-View-CH10B-01 *** */
SELECT          *
FROM            WorkPurchaseView
ORDER BY        CustomerLastName, CustomerFirstName, ArtistName;
```

The results of this query are shown in Figure 10B-89.

Triggers

Oracle triggers are PL/SQL or Java procedures that are invoked when a specified database activity occurs. Oracle supports a variety of different types of triggers. Some triggers are invoked on SQL commands that create new tables, views, or other database triggers. Other triggers are invoked once per SQL command, and still others are invoked for each row that is involved in the processing of an SQL command.

To understand the difference between the latter two trigger types, consider the following SQL update statement:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-UPDATE-CH10B-03 *** */
UPDATE      CUSTOMER
    SET         AreaCode = '425'
    WHERE       ZIPorPostalCode = '98119';
```

A command trigger will be fired once when the statement is processed. A row trigger will be fired once for every row that is updated during the processing of this statement. Row triggers are the most common, and we will consider only them in this chapter.

Oracle recognizes three types of row triggers: BEFORE, AFTER, and INSTEAD OF. BEFORE and AFTER triggers are placed on tables. INSTEAD OF triggers are placed on views. Each trigger type can be fired on INSERT, UPDATE, or DELETE commands.

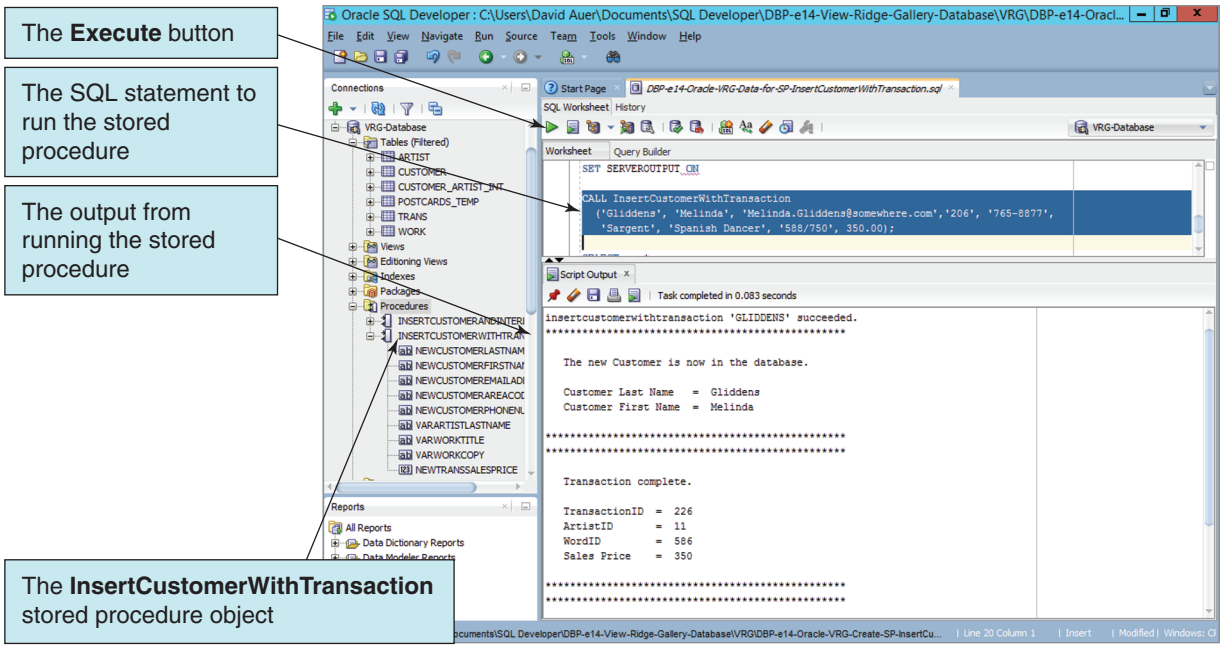


FIGURE 10B-83

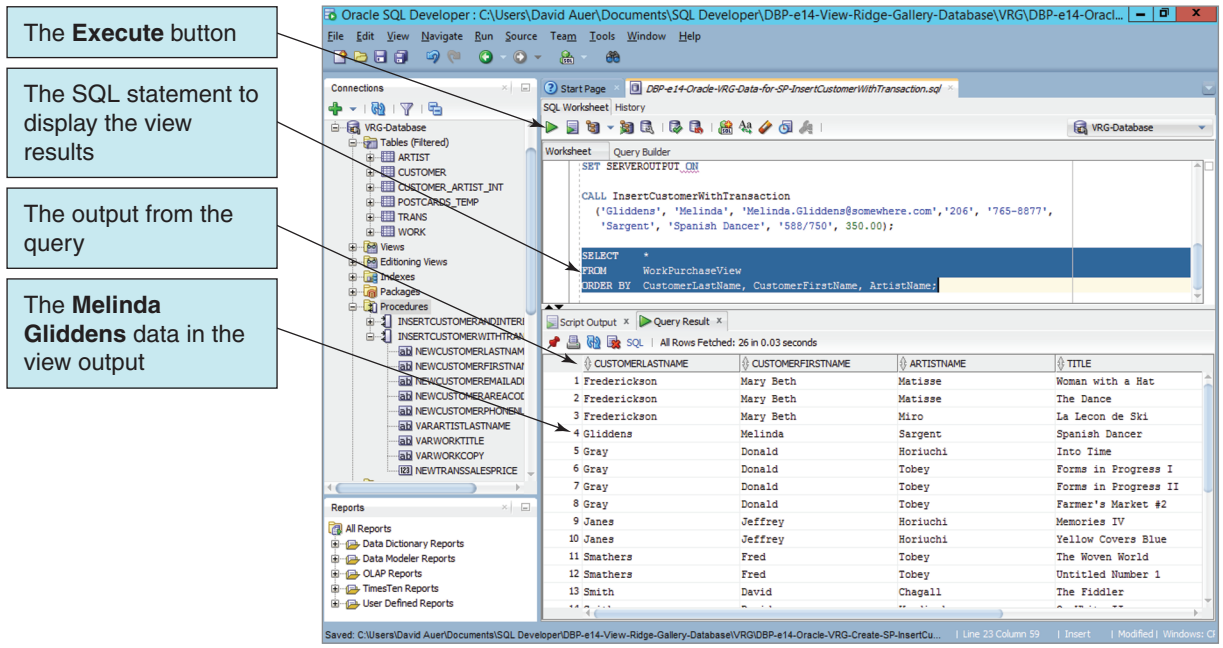
Running the InsertCustomerWithTransaction Stored Procedure

Because of the way that Oracle manages concurrency, AFTER triggers that update the table that caused the trigger to be fired can be problematic. For example, if table T1 has an AFTER UPDATE trigger, any code in the trigger that also attempts to process table T1 may not work correctly. When this occurs, Oracle issues a message like "Table T1 is mutating, trigger/function may not see it." For this reason, any actions that require processing the table that is firing the trigger are best done with BEFORE triggers.

AFTER triggers can be useful, however, when the action of the trigger applies to a table other than the one that fired the trigger. For example, if table T1 requires a child row in table

FIGURE 10B-84

The Inserted New Customer Data



T2, an AFTER trigger on T1 insert can be used to create the required T2 child. You will see an example of that use in Figure 10B-93.

The values of the columns of the table or the view upon which the trigger is based are available to the trigger. For insert and update triggers, the new values of the table or view columns can be accessed with the prefix *:new*. Thus, if table T1 has two columns, C1 and C2, when an insert or update trigger is fired on T1, the expression :new.C1 has the new value for the column C1 and the expression :new.C2 has the new value for the column C2.

For update and delete triggers, the old values of the table or view columns can be accessed with the prefix *:old*. Thus, :old.C1 will have the value of column C1 before the update or delete is processed.

In the next sections, we will discuss a trigger that computes a default value, one that enforces a data constraint, one that updates a view, and, finally, one that enforces a required child constraint.

A Trigger for Setting Default Values

Triggers can be used to set default values that are more complex than those that can be set with the default constraint on a column definition. For example, the View Ridge Gallery has a pricing policy that says that the default AskingPrice of a work of art depends on whether the art work has been in the gallery before. If not, the default AskingPrice is twice the AcquisitionPrice. If the work has been in the gallery before, the default price is the larger of twice the AcquisitionPrice or the AcquisitionPrice plus the average net gain of the work in the past. We would like to call this trigger the TRANS_BeforeInsertSetAskingPrice trigger, but this name is too long for Oracle Database, so we will call it the TRANS_BI_SetAskingPrice trigger. The PL/SQL code shown in Figure 10B-85 implements this pricing policy.

FIGURE 10B-85

The SQL Statements for the TRANS_BI_SetAskingPrice Trigger

```
CREATE OR REPLACE TRIGGER TRANS_BI_SetAskingPrice
BEFORE INSERT ON TRANS

FOR EACH ROW

DECLARE   varRowCount            Int;
          varWID                 Int;
          varTID                 Int;
          varAcquisitionPrice    Number(8,2);
          varNewAskingPrice      Number(8,2);
          varAvgNetProfit        Number(8,2);

BEGIN

   varTID := :new.TransactionID;
   varWID := :new.WorkID;
   varAcquisitionPrice := :new.AcquisitionPrice;

   -- First find if work has been here before.

   SELECT COUNT(*) INTO varRowCount
   FROM   TRANS
   WHERE  WorkID = varWID;

   -- Since this is an BEFORE trigger, varRowCount does not include the new row.

   IF (varRowCount = 0) THEN
      -- This is first time work has been in the gallery.
      -- Set varNewAskingPrice to twice the acquisition cost.
      varNewAskingPrice := (2 * varAcquisitionPrice);
      :new.AskingPrice := varNewAskingPrice;

   ELSIF (varRowCount > 0) THEN
```

```
    -- The work has been here before
    -- We have to determine the value of varNewAskingPrice
    BEGIN
        SELECT        AVG(NetProfit) INTO varAvgNetProfit
        FROM          ArtistWorkNetView AWNV
        WHERE         AWNV.WorkID = varWID
        GROUP BY      AWNV.WorkID;


        -- Now choose larger value for the new AskingPrice.
        IF ((varAcquisitionPrice + varAvgNetProfit)
               > (2 * varAcquisitionPrice)) THEN
            varNewAskingPrice := (varAcquisitionPrice + varAvgNetProfit);
            :new.AskingPrice := varNewAskingPrice;
        ELSE
            varNewAskingPrice := (2 * varAcquisitionPrice);
            :new.AskingPrice := varNewAskingPrice;
        END IF;
    END;
END IF;


-- The INSERT is completed. Print output
BEGIN
          DBMS_OUTPUT.PUT_LINE('************************************************');
          DBMS_OUTPUT.PUT_LINE('');
          DBMS_OUTPUT.PUT_LINE('   INSERT complete. ');
          DBMS_OUTPUT.PUT_LINE('');
          DBMS_OUTPUT.PUT_LINE('   TransactionID     =  '||varTID);
          DBMS_OUTPUT.PUT_LINE('   WorkID            =  '||varWID);
          DBMS_OUTPUT.PUT_LINE('   Acquisition Price =  '||varAcquisitionPrice);
          DBMS_OUTPUT.PUT_LINE('   Asking Price      =  '||varNewAskingPrice);
          DBMS_OUTPUT.PUT_LINE('');
          DBMS_OUTPUT.PUT_LINE('************************************************');
    END;
END;
/
```

FIGURE 10B-85

Continued

The BEFORE trigger in Figure 10B-85 uses the view ArtistWorkNetView, which was defined in Chapter 7 and which you should have created in the VRG database in the preceding section on views.

The trigger first counts the number of rows in TRANS having the :new value of WorkID. Because this is a BEFORE trigger, the work has not yet been added to the database, and the count will be zero if the work has not been in the gallery before. If this is the case, :new.AskingPrice is set to twice the AcquisitionPrice.

If the work has been in the gallery before, the average of NetPrice for this work is computed using the ArtistWorkNet view. Then the variable newPrice is computed as the sum of the average plus the acquisition price. Finally, :new.AskingPrice is set to the larger of newPrice or twice the AcquisitionPrice. Because this is a BEFORE trigger, the Avg built-in function can be used because the new row of WORK has not yet been added to the database and will not count in the average computation.

The computations in this trigger may be a problem, however, if either SalesPrice or AcquisitionPrice is null in any of the rows in the ArtistWorkView. The discussion of that problem, however, is beyond the scope of this chapter.

This trigger provides useful functionality for the gallery. It saves the View Ridge Gallery personnel considerable manual work in implementing their pricing policy and likely improves the accuracy of the results as well. If you are compiling this trigger in SQL Developer, note that sometimes when you compile a trigger using the **Execute Statement** button, you will get a strange window popping up labeled "Enter Binds". To avoid this, compile the trigger using the **Execute Script** button instead.

To test the trigger, we will begin by obtaining a new work for the View Ridge Gallery. Because Melinda Gliddens just bought the only copy of the print of John Singer Sargent's *Spanish Dancer*, we will replace it:

```
-- INSERT new art work.
/* *** SQL-INSERT-CH10B-04 *** */
INSERT INTO WORK VALUES(
     seqWID.NextVal,'Spanish Dancer', '635/750',
     'High Quality Limited Print',
     'American Realist style - From work in Spain', 11);
COMMIT;
-- Obtain the new WorkID
/* *** SQL-Query-CH10B-10 *** */
SELECT      WorkID
FROM          WORK
WHERE         ArtistID = 11
    AND     Title = 'Spanish Dancer'
    AND     Copy = '635/750';
```

The result SQL-Query-CH10B-10 gives us is the WorkID of the new art work, which in this case is 597:

	WORKID
1	597

We use this value in the SQL INSERT statement to record the new transaction:

```
-- Use the new WorkID value (597 in this case)
SET SERVEROUTPUT ON
/* *** SQL-INSERT-CH10B-05 *** */
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice, WorkID)
     VALUES (seqTID.NextVal, TO_DATE('11/08/2014','MM/DD/YYYY'),
     200.00, 597);
COMMIT;
```

Figure 10B-86 shows the results of the events triggered by the INSERT statement on TRANS. Note that the asking price for the new work (400.00) has been set to twice the acquisition cost (200.00), which is the correct value for a work that has not previously been in the gallery. This trigger provides useful functionality for the gallery. It saves the gallery personnel considerable manual work in implementing their pricing policy and likely improves the accuracy of the results as well.

A Trigger for Enforcing a Data Constraint

The View Ridge Gallery needs to track problem-customer accounts; these are customers either who have not paid promptly or who have presented other problems to the gallery. When a customer who is on the problem list attempts to make a purchase at the gallery, the gallery wants the transaction to be rolled back and a message displayed. Note that this feature requires an intertable CHECK constraint between the TRANS table and the CUSTOMER table, which, as we discussed in Chapter 7, requires a trigger.

To enforce this policy and the corresponding constraint, we need to add a column to the CUSTOMER table named isProblemAccount. This column will use the integer data type

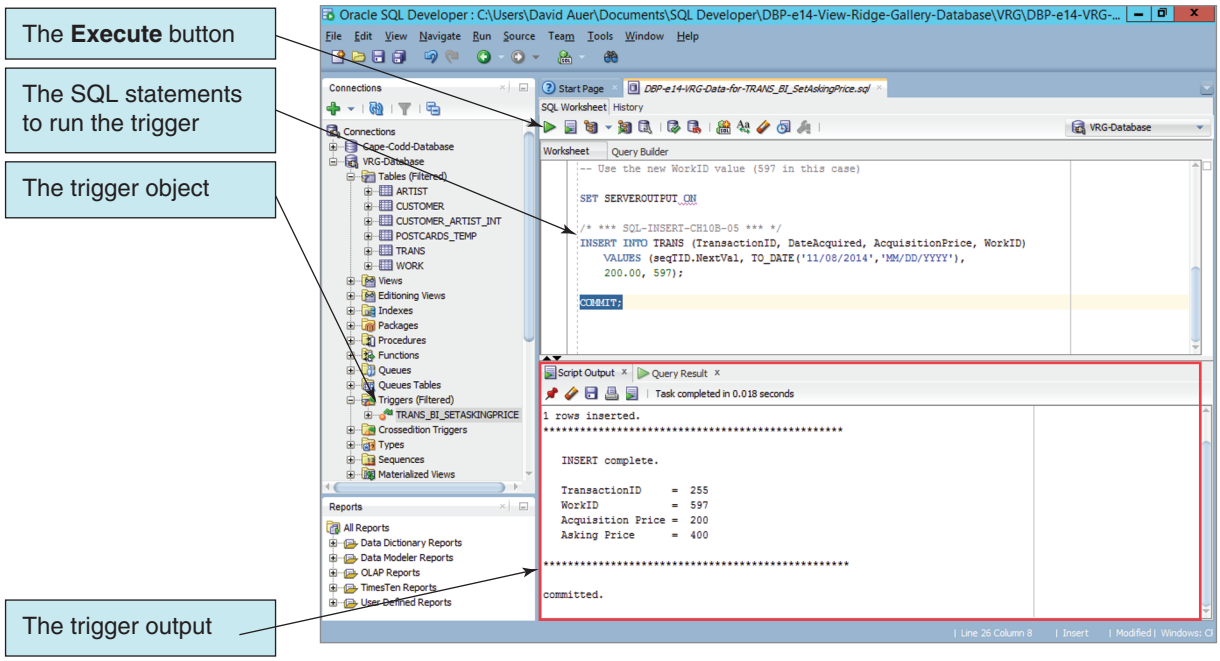


FIGURE 10B-86

Running the TRANS_BI_SetAskingPrice Trigger

and will have the possible values NULL, 0, and 1. Zero will indicate a good account; 1 will indicate a problem account. It looks like our new customer Melinda Gliddens had trouble with her previous payment, so we will set her isProblemAccount value to 1.

```
/* *** Add column isProblemAccount to CUSTOMER *** */
/* *** SQL-ALTER-TABLE-CH10B-02 *** */
ALTER TABLE CUSTOMER
    ADD     isProblemAccount        Int NULL;
/* *** Set intial column values for CUSTOMER.isProblemAccount *** */
/* *** SQL-UPDATE-CH10B-04 *** */
UPDATE      CUSTOMER
    SET     isProblemAccount = 0;
COMMIT;
/* *** Set column value for Melinda Gliddens *** */
/* *** SQL-UPDATE-CH10B-05 *** */
UPDATE      CUSTOMER
    SET     isProblemAccount = 1
    WHERE   LastName= 'Gliddens'
      AND   FirstName ='Melinda';
COMMIT;
/* *** Check CUSTOMER.isProblemAccount column values *** */
/* *** SQL-Query-CH10B-11 *** */
SELECT          CustomerID, LastName, FirstName, isProblemAccount
FROM            CUSTOMER
ORDER BY        CustomerID;
```

The results of the SELECT statement are:

	CUSTOMERID	LASTNAME	FIRSTNAME	ISPROBLEMACCOUNT
1	1000	Janes	Jeffrey	0
2	1001	Smith	David	0
3	1015	Twilight	Tiffany	0
4	1033	Smathers	Fred	0
5	1034	Frederickson	Mary Beth	0
6	1036	Warning	Selma	0
7	1037	Wu	Susan	0
8	1040	Gray	Donald	0
9	1041	Johnson	Lynda	0
10	1051	Wilkens	Chris	0
11	1052	Bench	Michael	0
12	1053	Gliddens	Melinda	1

Now we will create a trigger on TRANS named *TRANS_BU_IsProblemAccount*. With this trigger, when a customer makes a purchase, the trigger determines whether the customer is flagged by the value of the isProblemAccount data in the CUSTOMER table. If so, the triggering update statement (but not the entire transaction containing it) is rolled back and a message is displayed. The trigger code in Figure 10B-87 enforces this policy.

Note one interesting feature of the trigger code in Figure 10B-87. As noted there, this trigger will fire for every update on TRANS, including updates fired by another trigger, such as TRANS_BI_SetAskingPrice. But in that trigger, no customer is involved. Therefore, before completing the rest of this trigger, we need to make sure that there is actually a customer participating in a transaction whose account status needs to be checked. This is done by the trigger lines:

```
IF (varCID IS NULL) THEN RETURN;
END IF;
```

Note that we only want to exit the TRANS_BI_IsProblemAccount trigger if there is no customer, not rollback the transaction that fired the trigger. When writing multiple triggers, remember that they may be run from other actions besides the one that you originally created them to handle. The statement FOR EACH ROW causes this trigger to be a row trigger that is fired once for every TRANS row for which any column is updated.

OK, here comes Melinda to make another purchase—let's see what happens.

```
SET SERVEROUTPUT ON
/* *** SQL-UPDATE-CH10B-06 *** */
UPDATE      TRANS
    SET       DateSold = TO_DATE('11/18/2014', 'MM/DD/YYYY'),
              SalesPrice = 475.00,
              CustomerID = 1053
    WHERE     TransactionID = 229;
```

The resulting output is shown in Figure 10B-88. Looks like Melinda is off to talk to the manager about her account!

```
CREATE OR REPLACE TRIGGER TRANS_BU_IsProblemAccount
BEFORE UPDATE ON TRANS

FOR EACH ROW

DECLARE   varCID              Int;
          varTID              Int;
          varIsProblemAccount Int;

BEGIN
       varCID := :new.CustomerID;
       varTID := :new.TransactionID;

   /* This trigger will fire for every update of TRANS.
    * This includes updates without a Customer participating,
    * such as an update of AskingPrice using the
    * TRANS_AfterInsertSetAskingPrice trigger.
    * Therefore, make sure there is a Customer participating
    * in the Update of TRANS.
    */

   -- Check if Customer ID is NULL and if so RETURN.
   -- Do not ROLLBACK the transaction, just don't complete this trigger.

   IF (varCID IS NULL) THEN
      RETURN;
      END IF;

   -- Valid CustomerID.
   -- Obtain value of varIsProblemAcocunt.

   SELECT isProblemAccount INTO varIsProblemAccount
   FROM   CUSTOMER
   WHERE  CUSTOMER.CustomerID = varCID;

   IF (varIsProblemAccount = 1) THEN
      -- This is a problem account.
      -- Return (rollback not allowed in a trigger) and send message.
      BEGIN
         DBMS_OUTPUT.PUT_LINE('************************************************');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   Transaction canceled. ');
         DBMS_OUTPUT.PUT_LINE('   CustomerID   =  '||varCID);
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   Refer customer to the manager immediately. ');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('************************************************');
         RAISE_APPLICATION_ERROR(-20000,'Warning: Deadbeat Customer');
         RETURN;
      END;

ELSIF (varIsProblemAccount = 0) THEN
      -- This is a good account
      -- Let the transaction stand.
      BEGIN
         DBMS_OUTPUT.PUT_LINE('************************************************');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   Transaction completed. ');
         DBMS_OUTPUT.PUT_LINE('   Transaction ID   =  '||varTID);
         DBMS_OUTPUT.PUT_LINE('   CustomerID       =  '||varCID);
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('   Thank the customer for their business. ');
         DBMS_OUTPUT.PUT_LINE('');
         DBMS_OUTPUT.PUT_LINE('************************************************');
      END;
   END IF;
END;
/
```

FIGURE 10B-87

The SQL Statements for the TRANS_ BU_IsProblemAccount Trigger

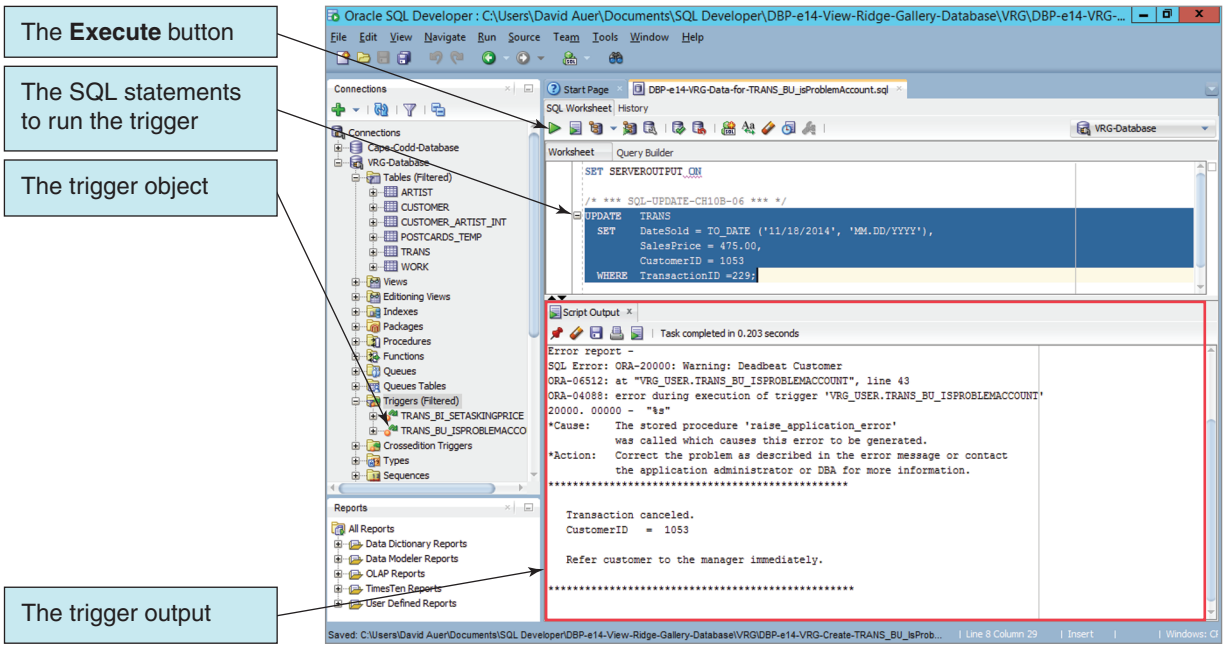


FIGURE 10B-88

Running the TRANS_BU_ IsProblemAccount Trigger

BY THE WAY

Using a table of valid or invalid values is more flexible and dynamic than placing such values in a CHECK constraint. For example, consider the CHECK constraint on Nationality values in the ARTIST table. If the gallery manager wants to expand the nationality of allowed artists, the manager will have to change the CHECK constraint using the ALTER TABLE statement. In reality, the gallery manager will have to hire a consultant to change this constraint.

A better approach is to place the allowed values of Nationality in a table, say, ALLOWED_NATIONALITY. Then write a trigger like that shown in Figure 10B-87 to enforce the constraint that new values of Nationality exist in ALLOWED_NATIONALITY. When the gallery owner wants to change the allowed artists, the manager would simply add or remove values in the ALLOWED_NATIONALITY table.

A Trigger for Updating a View

In Chapter 7, we discussed the problem of updating views. One such problem concerns updating views created via joins; it is normally not possible for the DBMS to know how to update tables that underlie the join. However, sometimes application-specific knowledge can be used to determine how to interpret a request to update a joined view.

Consider the view CustomerInterestsView shown in Figure 10B-60. It contains rows of CUSTOMER and ARTIST joined over their intersection table. CUSTOMER.LastName is given the alias CustomerLastName, CUSTOMER.FirstLastName is given the alias CustomerFirstName, and ARTIST.LastName is given the alias ArtistName.

A request to change the last name of a customer in CustomerInterests can be interpreted as a request to change the last name of the underlying CUSTOMER table. Such a request, however, can be processed only if the value of (CUSTOMER.LastName, CUSTOMER. FirstName) is unique. If not, the request cannot be processed.

The INSTEAD OF trigger shown in Figure 10B-89 implements this logic. First, the new and old values of the Customer columns in CustomerInterestsView are obtained. Then a correlated subquery is used to determine whether the old value of (CUSTOMER.LastName, CUSTOMER.FirstName) is unique. If so, the name can be changed, but otherwise no update can be made. For simplicity, we require both a new last name and a new first name even if only one of these (and it can be either one) is being changed.

```
CREATE OR REPLACE TRIGGER CIV_IO_ChangeCustomerName
INSTEAD OF UPDATE ON CustomerInterestsView

FOR EACH ROW

DECLARE   varRowCount                Int;
          varNewCustomerLastName     Char(25);
          varNewCustomerFirstName    Char(25);
          varOldCustomerLastName     Char(25);
          varOldCustomerFirstName    Char(25);

BEGIN

   -- Get values of new and old names.

   varNewCustomerLastName := :new.CustomerLastName;
   varNewCustomerFirstName := :new.CustomerFirstName;
   varOldCustomerLastName := :old.CustomerLastName;
   varOldCustomerFirstName := :old.CustomerFirstName;

   -- Count number of synonyms in CUSTOMER.
   SELECT    COUNT(*) INTO varRowCount
   FROM      CUSTOMER C1
   WHERE     C1.LastName = varOldCustomerLastName
      AND    C1.FirstName = varOldCustomerFirstName
      AND    EXISTS
             (SELECT   *
              FROM     CUSTOMER C2
              WHERE    C1.LastName = C2.LastName
                 AND   C1.FirstName = C2.FirstName
                 AND   C1.CustomerID <> C2.CustomerID);

   IF (varRowCount = 0) THEN
      -- The Customer name is unique.
      -- Update the Cusotmer record.
      BEGIN
         UPDATE    CUSTOMER
         SET       LastName = varNewCustomerLastName,
                   FirstName = varNewCustomerFirstName
         WHERE     LastName = varOldCustomerLastName
            AND    FirstName = varOldCustomerFirstName;

         -- Print update message.
   DBMS_OUTPUT.PUT_LINE('************************************************');
   DBMS_OUTPUT.PUT_LINE('');
   DBMS_OUTPUT.PUT_LINE('   The Customer name has been changed. ');
   DBMS_OUTPUT.PUT_LINE('');
   DBMS_OUTPUT.PUT_LINE('   Former Customer Last Name    = '||varOldCustomerLastName);
   DBMS_OUTPUT.PUT_LINE('   Former Customer First Name   = '||varOldCustomerFirstName);
   DBMS_OUTPUT.PUT_LINE('');
   DBMS_OUTPUT.PUT_LINE('   Updated Customer Last Name   = '||varNewCustomerLastName);
   DBMS_OUTPUT.PUT_LINE('   Updated Customer First Name  = '||varNewCustomerFirstName);
   DBMS_OUTPUT.PUT_LINE('');
   DBMS_OUTPUT.PUT_LINE('************************************************');
      END;
```

(*continued*)

FIGURE 10B-89

The SQL Statements for the CIV_IO_ChangeCustomerName Trigger

```
ELSIF (varRowCount > 0) THEN
       -- The Customer name is not unique.
       -- Return (rollback not allowed in a trigger) and send message.
       BEGIN
           DBMS_OUTPUT.PUT_LINE('************************************************');
           DBMS_OUTPUT.PUT_LINE('');
           DBMS_OUTPUT.PUT_LINE('    Transaction canceled. ');
           DBMS_OUTPUT.PUT_LINE('');
           DBMS_OUTPUT.PUT_LINE('    Customer Last Name    = '||varNewCustomerLastName);
           DBMS_OUTPUT.PUT_LINE('    Customer First Name   = '||varNewCustomerFirstName);
           DBMS_OUTPUT.PUT_LINE('');
           DBMS_OUTPUT.PUT_LINE('    The Customer name is not unique. ');
           DBMS_OUTPUT.PUT_LINE('');
           DBMS_OUTPUT.PUT_LINE('************************************************');
       RETURN;
       END;
   END IF;
END;
/
```

FIGURE 10B-89
Continued

This trigger needs to be tested against cases in which the customer name is unique and cases in which the customer name is not unique. Figure 10B-90 shows the case in which the customer name is unique. View Ridge Gallery's two newest customers, Michael Bench and Melinda Gliddens, just got married after meeting at a View Ridge gallery opening, and Melinda wants us to change her last name. The SQL statement to do this is:

```
SET SERVEROUTPUT ON
/* *** SQL-UPDATE-CH10B-07 *** */
UPDATE       CustomerInterestsView
    SET          CustomerLastName = 'Bench',
                 CustomerFirstName = 'Melinda'
    WHERE        CustomerLastName = 'Gliddens'
        AND      CustomerFirstName = 'Melinda';
```

FIGURE 10B-90
Running the CIV_IO_ChangeCustomerName Trigger

Note that the UPDATE command was issued against the view. As indicated in the Messages pane, Melinda is now Melinda Bench.

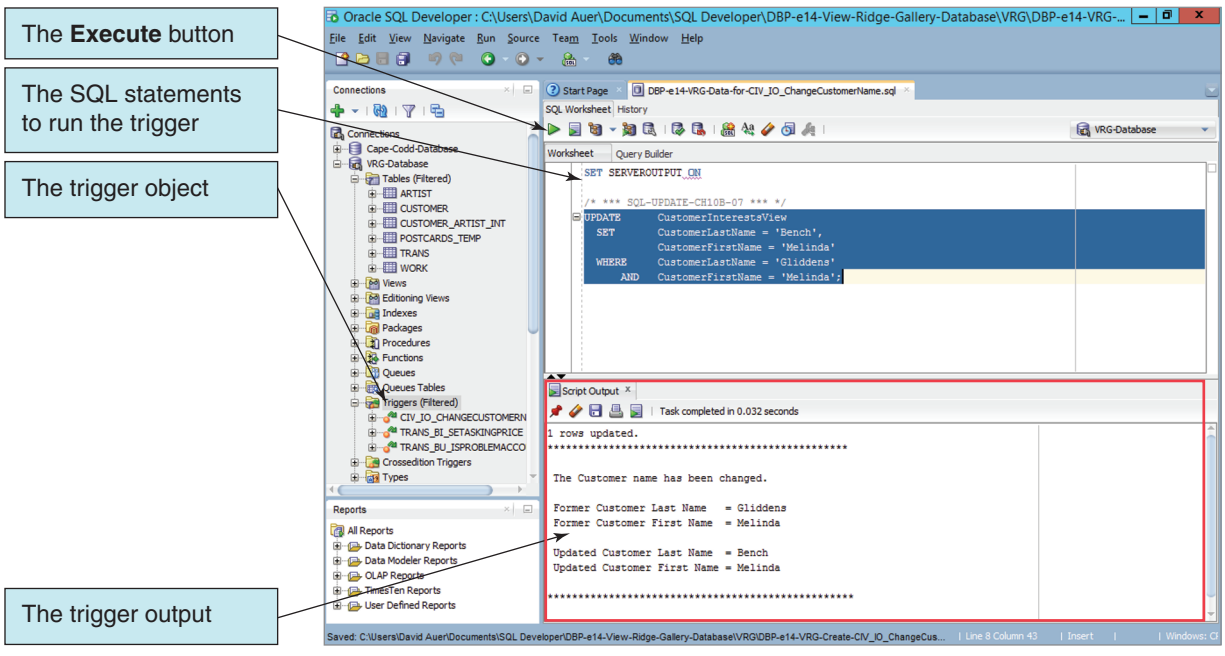


FIGURE 10B-91

Actions to Enforce Minimum Cardinality for the WORK-to-TRANS Relationship

WORK Is Required Parent TRANS Is Required Child	Action on WORK (Parent)	Action on TRANS (Child)
Insert	Create a TRANS row	New TRANS must have a valid WorkID (enforced by DBMS).
Modify key or foreign key	Prohibit—WORK uses a surrogate key	Prohibit—TRANS uses a surrogate key, and TRANS cannot change to a different WORK.
Delete	Prohibit—Cannot delete a WORK with TRANS children (enforced by DBMS by lack of CASCADE DELETE)	Cannot delete the last child [Actually, data related to a transaction is never deleted (business rule)].

A Trigger for Enforcing a Required Child Constraint

The VRG database design includes an M-M relationship between WORK and TRANS. Every WORK must have a TRANS to store the price of the work and the date the work was acquired, and every TRANS must relate to a WORK parent. Figure 10B-96 shows the tasks that must be accomplished to enforce this constraint; it is based on the boilerplate shown in Figure 6-29(b).

Because the CREATE TABLE statement for TRANS in Figure 10B-48 defines TRANS.WorkID as NOT NULL and defines the FOREIGN KEY constraint without cascading deletions, the DBMS will ensure that every TRANS has a WORK parent. So, we need not be concerned with enforcing the insert on TRANS or the deletion on WORK. As stated in Figure 10B-91, the DBMS will do that for us. Also, we need not be concerned with updates to WORK.WorkID because it is a surrogate key (if need be, we can use a trigger to deny such updates to a surrogate primary key).

Three constraints remain that must be enforced by triggers: (1) ensuring that a TRANS row is created when a new WORK is created; (2) ensuring that TRANS.WorkID never changes; and (3) ensuring that the last TRANS child for WORK is never deleted.

We can enforce the second constraint by writing a trigger on the update of TRANS that checks for a change in WorkID. If there is such a change, the trigger can prevent the change. Concerning the third constraint, the gallery has a business policy that no TRANS data ever be deleted. Thus, we not only need to disallow the deletion of the last child, we also need to disallow the deletion of any child. We can do this by writing a trigger on the deletion of TRANS that prevents any attempted deletion–if the gallery allowed TRANS deletions, we could enforce the deletion constraint using views, as shown in Chapter 7, Figures 7-28 and 7-29. The triggers for enforcing the second and third constraints are simple, and we leave them as exercises in Review Questions 10B.52 and 10B.53.

However, the first constraint is a problem. We could write a trigger on the WORK INSERT to create a default TRANS row, but this trigger will be called before the application has a chance to create the TRANS row itself. The trigger would create a TRANS row, and then the application may create a second one. To guard against the duplicate, we could then write a trigger on TRANS to remove the row the WORK trigger created in those cases when the application creates its own trigger. However, this solution is awkward at best.

A better design is to require the applications to create the WORK and TRANS combination via a view. For example, consider the view WorkAndTransView:

```
/* *** SQL-CREATE-VIEW-CH10B-02*** */
CREATE VIEW WorkAndTransView AS
  SELECT  Title, Copy, Medium, Description, ArtistID,
          DateAcquired, AcquisitionPrice
  FROM    WORK W JOIN TRANS T
    ON    W.WorkID = T.WorkID;
```

The DBMS will not be able to process an insert on this view. We can, however, define an INSTEAD OF trigger to process the insert. Our trigger, named *WATV_IO_InsertWorkWithTrans*, will create both a new row in WORK and the new required child in TRANS. The code for this trigger is shown in Figure 10B-92. Note that with this solution, applications must not be allowed to insert WORK rows directly. They must always insert them via the view WorkAndTransView.

To test our trigger, we will add a new work to the VRG database. Melinda, now Mrs. Bench, has worked out her account problems with the View Ridge Gallery and has completed her purchase of the print of Horiuchi's *Color Floating in Time*.

```
/* *** SQL-UPDATE-CH10B-08 *** */
UPDATE      CUSTOMER
    SET         isProblemAccount = 0
    WHERE       LastName = 'Bench'
      AND       FirstName = 'Melinda';
COMMIT;
```

FIGURE 10B-92

The SQL Statements for the WATV_IO_InsertWorkWithTrans Trigger

```
CREATE OR REPLACE TRIGGER WATV_IO_InsertWorkWithTrans
INSTEAD OF INSERT ON WorkAndTransView

FOR EACH ROW

DECLARE   varRowCount          Int;
          varTID               Int;
          varWID               Int;
          varTitle             Char(35);
          varCopy              Char(12);
          varMedium            Char(35);
          varDescription       Varchar(1000);
          varAID               Int;
          varDateAcquired      Date;
          varAcquisitionPrice  Number(8,2);
          varAskingPrice       Number(8,2);

BEGIN

   -- Get available values from Insert on the view.
   varTitle := :new.Title;
   varCopy := :new.Copy;
   varMedium := :new.Medium;
   varDescription := :new.Description;
   varAID := :new.ArtistID;
   varDateAcquired := :new.DateAcquired;
   varAcquisitionPrice := :new.AcquisitionPrice;

   -- Insert new row into WORK.
   INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
      VALUES(
      seqWID.NextVal, varTitle, varCopy, varMedium, varDescription, varAID);

   -- Get new WorkID surrogate key value using {Seq}.CurrVal function.
   varWID := seqWID.CurrVal;

   -- Insert new row into TRANS.
   -- Note that INSERT will trigger TRANS_AI_SetAskingPrice.
   INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice, WorkID)
      VALUES(seqTID.NextVal, varDateAcquired, varAcquisitionPrice, varWID);

   -- Get new TranasctionID surrogate key value using {Seq}.CurrVal function.
   varTID := seqTID.CurrVal;

   -- Get new AskingPrice set by TRANS_AI_AskingPrice.
   SELECT    AskingPrice INTO varAskingPrice
   FROM      TRANS
   WHERE     TransactionID = varTID;
```

```
    -- Print results message.
    DBMS_OUTPUT.PUT_LINE('************************************************');
    DBMS_OUTPUT.PUT_LINE('');
    DBMS_OUTPUT.PUT_LINE('   The new work has been inserted into WORK and TRANS.');
    DBMS_OUTPUT.PUT_LINE('');
    DBMS_OUTPUT.PUT_LINE('   TransactionID     = '||varTID);
    DBMS_OUTPUT.PUT_LINE('   WorkID            = '||varWID);
    DBMS_OUTPUT.PUT_LINE('   ArtistID          = '||varAID);
    DBMS_OUTPUT.PUT_LINE('   Title             = '||varTitle);
    DBMS_OUTPUT.PUT_LINE('   Copy              = '||varCopy);
    DBMS_OUTPUT.PUT_LINE('   Medium            = '||varMedium);
    DBMS_OUTPUT.PUT_LINE('   Description       = '||varDescription);
    DBMS_OUTPUT.PUT_LINE('   DateAcquired      = '||varDateAcquired);
    DBMS_OUTPUT.PUT_LINE('   Acquisition Price = '||varAcquisitionPrice);
    DBMS_OUTPUT.PUT_LINE('   Asking Price      = '||varAskingPrice);
    DBMS_OUTPUT.PUT_LINE('************************************************');
END;
/
```

FIGURE 10B-92
Continued

```
/* *** SQL-UPDATE-CH10B-09 *** */
UPDATE      TRANS
    SET       DateSold = TO_DATE('11/18/2014', 'MM/DD/YYYY'),
              SalesPrice = 475.00,
              CustomerID = 1053
    WHERE     TransactionID = 229;
COMMIT;
```

Therefore, we will restock a copy of this print into the gallery.

```
SET SERVEROUTPUT ON
/* *** SQL-INSERT-CH10B-06 *** */
INSERT INTO WorkAndTransView
     VALUES('Color Floating in Time', '493/750',
     'High Quality Limited Print',
     'Northwest School Abstract Expressionist style', 18,
     TO_DATE('02/05/15', 'MM/DD/YYYY'), 250.00);
```

We run the transaction, as shown in Figure 10B-93. Note that the WATV_IO_InsertWorkWithTrans trigger actually sets off two other triggers. First, it fires the INSERT trigger TRANS_BI_SetAskingPrice, which then fires the UPDATE trigger TRANS_BU_IsProblemAccount. Because no customer is involved in this transaction, the TRANS_BU_IsProblemAccount trigger returns control without completing the trigger (see the previous discussion of the code for this trigger). However, the TRANS_BI_SetAskingPrice trigger does run and sets the new asking price for the new work. Therefore, the two triggers generate the output shown in Figure 10B-93, and the new work will now have an asking price.

Exception Handling

This discussion has omitted a discussion of PL/SQL exception handling. This is unfortunate because exception handling is both important and useful. There's just too much to do. If you program in PL/SQL in the future, however, be sure to learn about this important topic. It can be used in all types of PL/SQL programming, but it is especially useful in BEFORE and INSTEAD OF triggers for canceling pending updates. Exceptions are necessary because transactions in Oracle Database cannot be rolled back in triggers. Exceptions can be used to generate error and warning messages. They also keep users better informed about what the trigger has done.

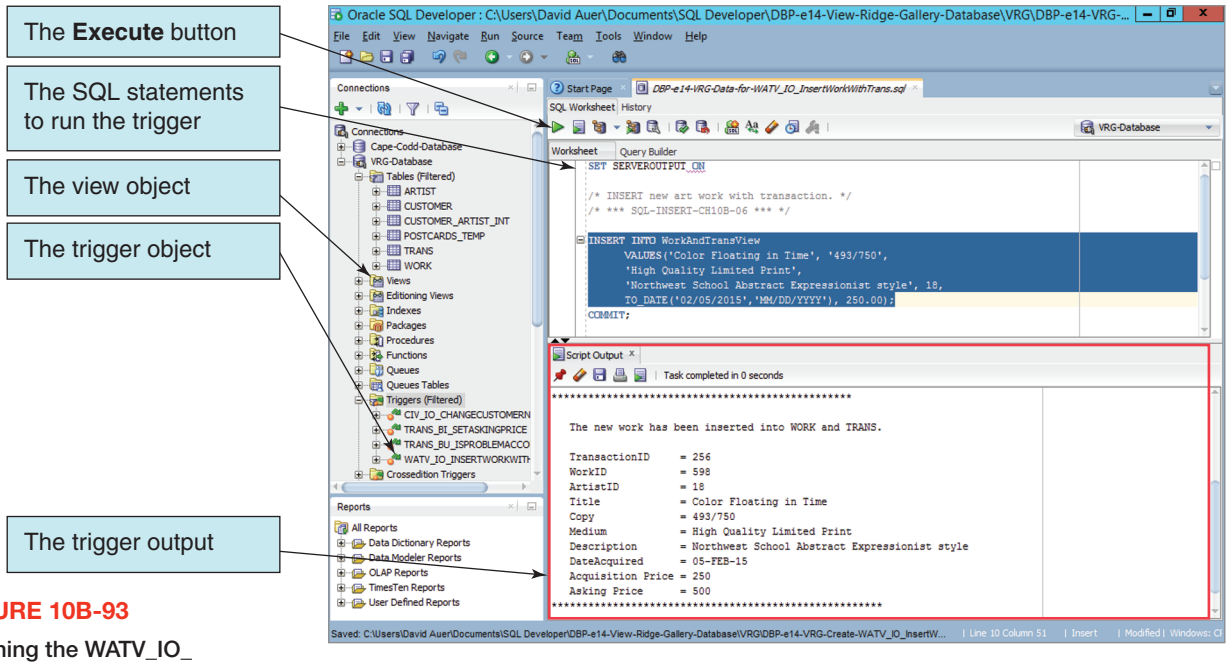


FIGURE 10B-93

Running the WATV_IO_InsertTransWithWork Trigger

Oracle Database Concurrency Control

Concurrency control in general is described and discussed in Chapter 9. Oracle Database supports three different transaction isolation levels and also allows applications to place locks explicitly. Explicit locking is not recommended, however, because such locking can interfere with Oracle Database's default locking behavior and because it increases the likelihood of transaction deadlock.

The Oracle Database change management and locking design is ingenious and sophisticated. With it, Oracle Database never makes dirty reads; it reads only committed changes. Oracle Database supports read-committed, serializable, and read-only transaction isolation levels. The first two are defined in the 1992 ANSI standard; read only is unique to Oracle Database. Figure 10B-94 summarizes these isolation levels.

Before discussing the implementation of transaction isolation levels, you need to understand how Oracle Database processes database changes. Oracle Database maintains a **System Change Number (SCN)**, which is a database-wide value that is incremented by Oracle Database whenever database changes are made. When a row is changed, the before image of the row is placed in a **rollback segment**, which is a section of memory maintained by Oracle Database. The before image includes the SCN that was in the row prior to the change. Then the row is changed, and Oracle Database increments the SCN and places the new SCN value in the changed row. When an application issues an SQL statement like:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-UPDATE-CH10B-10 *** */
UPDATE      MYTABLE
    SET         MyColumn1 = 'NewValue'
    WHERE       MyColumn2 = 'Something';
```

the value of SCN that was current at the time the statement started is recorded. Call this value the *Statement SCN*. While processing the query, in this case while looking for rows with MyColumn2 = 'Something', Oracle Database selects only rows that have committed changes with an SCN value less than or equal to the Statement SCN. When it finds a row with a committed change and SCN value greater than the Statement SCN, it looks in the rollback

FIGURE 10B-94
Oracle Transaction Isolation

Read Committed	The default Oracle isolation level. Dirty reads are not possible, but repeated reads may yield different data. Phantoms are possible. Each statement reads consistent data. When blocked for updates, statements are rolled back and restarted when necessary. Deadlock is detected and one of the blocking statements is rolled back.
Serializable	Dirty reads are not possible, repeated reads yield the same results, and phantoms are not possible. All statements in the transaction read consistent data. "Cannot serialize" error occurs when a transaction attempts to update or delete a row with a committed data change that occurred after the transaction started. Also occurs when blocking transactions or statements commit their changes or when the transaction is rolled back due to deadlock. Application programs need to be written to handle the "Cannot serialize" exception.
Read Only	All statements read consistent data. No inserts, updates, or deletions are possible.
Explicit Locks	Not recommended.

segment to find an earlier version of the row. It searches the rollback segments until it finds a version of the row with a committed change having an SCN less than the Statement SCN.

In this way, SQL statements always read a consistent set of values—those that were committed at or before the time the statement was started. As you will see, this strategy is sometimes extended to apply to transactions. In that case, all of the statements in a transaction read rows having an SCN value less than the SCN that was current when the transaction started. Again, this design means that Oracle Database reads only committed changes—dirty reads are *not* possible.

Read-Committed Transaction Isolation Level

Recall from Chapter 9 that dirty reads are not allowed with read-committed isolation, but reads may not be repeatable and phantoms are possible. Because Oracle Database's design prohibits reading dirty data, read committed is Oracle Database's default transaction isolation level. Because of the way Oracle Database implements read-committed isolation, each SQL statement is consistent, but two different SQL statements in the same transaction may read inconsistent data. If transaction-level consistency is required, serializable isolation must be used. Do not confuse statement consistency with the lost update problem, however. Oracle Database prohibits lost updates because it never reads dirty data.

Because of the way it uses the SCN, Oracle Database never needs to place read locks. When a row is to be changed or deleted, however, Oracle Database places an exclusive lock on the row before making the change or deletion. If another transaction has an exclusive lock on the row, the statement waits. If the blocking transaction rolls back, the change or deletion proceeds.

If the blocking transaction commits, the new SCN value is given to the statement, and the statement (not the transaction) rolls back and starts over. When a statement is rolled back, changes already made by the statement are removed using the rollback segments.

Because exclusive locks are used, deadlock can occur. When that happens, Oracle Database detects the deadlock using a wait-for graph and rolls back one of the conflicting statements.

Serializable Transaction Isolation Level

As you learned in Chapter 9, with serializable transaction isolation, dirty reads are not possible, reads are always repeatable, and phantoms cannot occur. Oracle Database supports serializable transaction isolation, but the application program must play a role for it to work.

Use the SET command to change the transaction isolation level. The following statement establishes serializable isolation for the duration of a transaction:

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

To change the isolation level for all transactions, use the ALTER command:

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
ALTER SESSION SET ISOLATION_LEVEL SERIALIZABLE;
```

When the isolation level is serializable, Oracle Database saves the SCN at the time the transaction started. Call this value the Transaction SCN. As the transaction proceeds, Oracle reads only committed changes that have an SCN value less than or equal to the Transaction SCN. Hence, reads are always repeatable and phantoms are not possible.

As long as the transaction does not attempt to update or delete any row having a committed change with an SCN greater than the Transaction SCN, the transaction proceeds normally. If, however, the transaction does attempt to update or delete such a row, Oracle Database issues a "Cannot serialize" error when the update or delete occurs. At that point, the application program must play a role. It can commit changes made to that point, roll back the entire transaction, or take some other action. Any program that executes under serializable isolation must include such exception-handling code.

Also, when a transaction running under serializable isolation attempts to update or delete a row that has been locked exclusively by a different transaction or statement, the transaction waits. If the blocking transaction or statement later rolls back, the transaction can continue. If, however, the blocking transaction commits, Oracle Database generates the "Cannot serialize" error, and the application needs to process that exception. Similarly, if a serializable transaction is rolled back due to deadlock, the "Cannot serialize" error is also generated.

Read-Only Transaction Isolation

With this isolation level, the transaction reads only rows having committed changes with an SCN value less than or equal to the Transaction SCN. If the transaction encounters rows with committed changes having an SCN value greater than the Transaction SCN, Oracle Database searches the rollback segments and reconstructs the row as it was prior to the Transaction SCN. With this level of transaction isolation, no inserts, updates, or deletions are allowed.

Additional Locking Comments

The application can invoke locks explicitly using the SELECT FOR UPDATE form of the SELECT statement. This is not recommended, and you should not use it until you have learned much more about Oracle locking than we have described here.

Behind the scenes, Oracle Database uses quite a wide variety of locks to provide isolation levels. Oracle Database has a row-share lock as well as several different types of table locks. Other locks are used internally within Oracle Database. You can learn more about these locks in the Oracle documentation.

To reduce the likelihood of lock conflict, Oracle Database does not promote locks from one level to another. Row locks remain row locks, even if there are hundreds of them on hundreds of rows of a table. Oracle Corporation claims that not promoting locks is an advantage, and it probably is—especially given the rest of the Oracle Database lock architecture.

Oracle Database Backup and Recovery

Oracle Database provides a sophisticated set of facilities and utilities for backup and recovery processing. They can be used in many different ways to provide appropriate backup and recovery for databases, ranging from a small workgroup database that can be backed up when

it is unused at night to large interorganizational databases that must be operational 24 hours per day, 7 days per week (24/7) and can never be shut down.

Oracle Database Recovery Facilities

Oracle Database maintains three types of files that are important for backup and recovery: datafiles, ReDo files, and control files.

Datafiles, which we discussed in conjunction with tablespaces, contain user and system data. Because of the way that Oracle Database writes data to disk, the datafiles may contain both committed and uncommitted changes at any moment in time. Of course, Oracle Database processes transactions so these uncommitted changes are eventually either committed or removed, but a snapshot of the datafiles at any arbitrary moment includes uncommitted changes. Thus, when Oracle Database shuts down or when certain types of backups are made, the datafiles must be cleaned up so only committed changes remain in them.

ReDo files contain logs of database changes; they are backups of the rollback segments used for concurrent processing. There are two types of ReDo files. **OnLine ReDo files** are maintained on disk and contain the rollback segments from recent database changes. **Offline ReDo files**, or **Archive ReDo files**, are backups of the OnLine ReDo files. They are stored separately from the OnLine ReDo files and need not necessarily reside on disk media. Oracle Database can operate in either ARCHIVELOG or NOARCHIVELOG mode. If it is running in ARCHIVELOG mode, when the OnLine ReDo files fill up, they are copied to the Archive ReDo files.

Control files are small files that store the SCN and the name, contents, and locations of various files used by Oracle Database. Control files are frequently updated by Oracle Database, and they must be available for a database to be operational.

Control files and OnLine ReDo files are so important that Oracle Database recommends that two active copies of them be kept, a process called **multiplexing** in Oracle Database terminology.

Types of Failure

Oracle Database recovery techniques depend on the type of failure. When an **application failure** occurs–because of application logic errors, for instance–Oracle Database simply rolls back uncommitted changes made by that application using the in-memory rollback segments and OnLine ReDo files as necessary.

Other types of failure recovery are more complicated and depend on the failure type. An **instance failure** occurs when Oracle Database itself fails due to an operating system or computer hardware failure. A **media failure** occurs when Oracle Database is unable to write to a physical file. This may occur because of a disk head crash or other disk failure because needed devices are not powered on or because a file is corrupt.

Database Instance Failure Recovery

When Oracle Database is restarted after a database instance failure, it looks first to the control file to find out where all the other files are located. Then it processes the OnLine ReDo logs against the datafiles. It rolls forward all changes in the ReDo log that were not yet written to the datafiles at the time of failure. In the process of rolling forward, rollback segments are filled with records of transactions in the ReDo log.

After rollforward, the datafiles may contain uncommitted changes. These uncommitted changes could have been in the datafiles at the time of the instance failure, or they could have been introduced by rollforward. Either way, Oracle Database eliminates them by rolling back such uncommitted changes using the rollback segments that were created during rollforward. So transactions do not need to wait for the rollback to complete, all uncommitted transactions are marked as DEAD. If a new transaction is blocked by a change made by a DEAD transaction, the locking manager destroys the locks held by the DEAD transaction.

The Archive ReDo logs are not used for instance recovery. Accordingly, instance recovery can be done in either ARCHIVELOG or NOARCHIVELOG mode.

Media Failure Recovery

To recover from a media failure, the database is restored from a backup. If the database was running in NOARCHIVELOG, nothing else can be done. The OnLine ReDo log is not useful

because it concerns changes made long after the backup was made. The organization must find another way to recover changes to the database. (This would be the wrong time to start thinking about this, by the way.)

If Oracle Database was operating in ARCHIVELOG mode, the OnLine ReDo logs will have been copied to the archive. To recover, the database is restored from a backup and is rolled forward by applying Archive ReDo log files. After this rollforward finishes, changes made by uncommitted transactions are removed by rolling them back, as described previously.

Two kinds of backups are possible. A **consistent backup** is one in which all uncommitted changes have been removed from the datafiles. Database activity must be stopped, all buffers must be flushed to disk, and changes made by any uncommitted transactions removed. Clearly, this type of backup cannot be done if the database supports 24/7 operations.

An **inconsistent backup** may contain uncommitted changes. It is a sort of flying backup that is made while Oracle Database is processing the database. For recovery, such backups can be made consistent by applying the archive log records to commit or roll back all transactions that were in process when the backup was made. Inconsistent backups can be made on portions of the database. For example, in a 24/7 application, one-seventh of the database can be backed up every night. Over a week's time, a copy of the entire database will have been made.

The Oracle Recovery Manager (RMAN) is a utility used to create backups and to perform recovery. RMAN can be instructed to create a special recovery database that contains data about recovery files and operations. The specifics of this program are beyond the scope of this discussion, but more information can be found at *www.oracle.com/technetwork/database/features/availability/rman-overview-096633.html*.

Topics Not Discussed in This Chapter

Several important Oracle Database features were not discussed in this chapter. For one, Oracle Database supports object-oriented structures, and developers can use them to define their own abstract data types. Oracle Database can also be used to create and process databases that are hybrids of traditional databases and object databases. Such hybrids, called *object-relational databases*, have not received strong market endorsement, and we will not consider them in this text.

Enterprise Oracle Database also supports distributed database processing, whereby the database is stored on more than one computer. Additionally, there are many Oracle Database utilities that we have not discussed. The Oracle Database Loader is a utility for inputting bulk data into an Oracle Database database. Other utilities can be used to measure and tune Oracle Database performance.

We have, however, discussed the most important Oracle Database features and topics here. If you have understood these concepts, you are well on your way to becoming a successful Oracle Database developer.

Summary

Oracle Database is a powerful and robust DBMS that runs on many different operating systems and has many different products. This chapter addresses the use of the Oracle Database SQL Developer utility, which can be used to create and process SQL and PL/SQL with all versions of Oracle Database. PL/SQL is a language that adds programming facilities to the SQL language.

You can create a database using the Database Configuration Assistant, the Oracle-supplied database creation procedures, or the SQL CREATE DATABASE command. The Database Configuration Assistant creates default database and log files.

The Web-based Enterprise Manager Database Control utility is used for administration of an Oracle Database

database instance. The Enterprise Manager is used to create and manage namespaces, datafiles, user accounts, and roles. It is used to manage database instance security using roles, system privileges, and object privileges.

Oracle Database security components include User accounts, Profiles, System Privileges, Object Privileges, and Roles. A User account has a Profile that specifies resource limits on the User and handles password management. A System privilege is the right to perform a task on an Oracle Database resource. An Object privilege is the right to perform a task on a specific Oracle Database object such as a specific table or view. Roles can be assigned to Users and consist of groups of System privileges, Object privileges, and other Roles. A User has all privileges that have been assigned directly to the user account plus all of the privileges of all assigned Roles and all Roles that are inherited through Roles' connections.

When an Oracle Database user is created, an Oracle Database schema is created at the same time and with the same name. The schema is a logical container, which allows the user to see all the objects that he or she has permissions to work with.

Oracle SQL*Plus is a command-line utility for Oracle Database database administration and application development. It is a favorite of many Oracle Database users, but there is now a GUI utility named SQL Developer that is much easier to use for application development. SQL Developer has an excellent text editor, and the utility can be used to manage structures, such as tables and views, and to manage user accounts, passwords, roles, and privileges.

PL/SQL statements and Java programs can be placed in the database as user-defined functions and stored procedures and invoked from other PL/SQL programs or from application programs. Two examples of stored procedures are shown in the chapter. Oracle Database triggers are PL/SQL or Java programs that are invoked when a specified database activity occurs. Examples of BEFORE, AFTER, and INSTEAD OF triggers are shown in the chapter.

Oracle Database supports read-committed, serializable, and read-only transaction isolation levels. Because of the way SCN values are processed, Oracle Database never reads dirty data. Serializable isolation is possible, but the application program must be written to process the "Cannot serialize" exception. Applications can place locks explicitly using SELECT FOR UPDATE commands, but this is not recommended.

Three types of files are used in Oracle Database recovery: datafiles, ReDo log files, and control files. If running in ARCHIVELOG mode, Oracle Database logs all changes to the database. Oracle Database can recover from application failure and instance failure without using the archived log file. Archive logs are required, however, to recover from media failure. Backups can be consistent or inconsistent. An inconsistent backup can be made consistent by processing an archive log file.

Key Terms

/* and */ signs
Adobe Flash Player
application failure
Archive ReDo files
ARCHIVELOG mode
BEGIN…END block
CLOSE keyword
command-line utility
Commit Transaction button
Container Database (CDB)
Connect As
consistent backup
control files
control-of-flow statements
Database Configuration Assistant
database instance
datafiles
DECLARE CURSOR keywords
delimited identifier
EM Express
Enterprise Manager
EXIT WHEN keywords
extract, transform, and load (ETL)
FETCH keyword
FOR keyword
IF…THEN…ELSE…END IF keywords
inconsistent backup
index
instance failure
INSTR
integrated development environment (IDE)
Java
Java Development Kit (JDK)
Java Runtime Environment (JRE)
JavaScript
LOOP keyword
media failure
multiplexing
NetBeans
NOARCHIVELOG mode
Offline ReDo files
OnLine ReDo files
OPEN keyword
Oracle Application Express workspace
Oracle Database 11g Express Edition Install Wizard
Oracle Database 12*c* (Oracle 12*c*)
Oracle Database Configuration Assistant (DBCA)
Oracle Database Express Edition 11*g* Release 2
Oracle Database SYS system account
Oracle Database TO_DATE function
Oracle Database XE
Oracle Database XE 11.2
Oracle Enterprise Manager Database Express 12c
Oracle SQL Developer
Oracle Universal Installer (OUI)
parameter
Pluggable Database (PDB)
Procedual Language/SQL (PL/SQL)
ReDo files
reserved word
RETURN keyword
Rollback Transaction button
rollback segment

schema
sequence
software development kit (SDK)
spreadsheet
SQL ALTER TABLE statement
SQL COMMIT statement
SQL CREATE DATABASE command
SQL CREATE FUNCTION statement
SQL CREATE PROCEDURE statement
SQL CREATE TRIGGER statement
SQL DROP PROCEDURE statement
SQL ROLLBACK statement
SQL script comments
SQL scripts
SQL SET TRANSACTION statement
SQL*Plus
stored function
SUBSTR
System Change Number (SCN)
System Identifier (SID)
tablespace
two dashes (- -)
user-defined function
variable
WHILE keyword
worksheet

Review Questions

10B.1 Describe the general characteristics of Oracle Database 12*c* and the Oracle Database 12*c* product suite. Explain why these characteristics mean there is considerable complexity to master.

10B.2 What is the Oracle Universal Installer (OUI), and what is its purpose?

10B.3 What is the Oracle Database Configuration Assistant (DBCA), and what is its purpose?

10B.4 What is the Enterprise Manager 12*c* Database Express, and what is its purpose? What is the difference between a *Container Database (CDB)* and a *Pluggable Database (PDB)?*

10B.5 What is the Oracle Express XE 11.2 utility, and what is its purpose?

10B.6 What is SQL*Plus, and what is its purpose? What is SQL Developer, and what is its purpose?

10B.7 Name two ways of creating an Oracle Database database. Which is the easiest?

10B.8 What is an Oracle Database database instance?

10B.9 What is an Oracle Database tablespace? What is the purpose of a tablespace? What are some standard tablespaces?

10B.10 What is an Oracle Database datafile? How is it related to a tablespace?

10B.11 Explain the use of User account, Profiles, System privileges, Object privileges, and Roles in Oracle Database security.

10B.12 Show the SQL statement necessary to create a table named T1 with columns C1, C2, and C3. Assume that C1 is a surrogate key. Assume that C2 has character data of maximum length 50 and that C3 contains a date.

10B.13 Show the statement necessary to create a sequence starting at 50 and incremented by 2. Name your sequence *T1Seq*.

10B.14 Show how to insert a row into table T1 (Review Question 10B.12) using the sequence created in Review Question 10B.13.

10B.15 Show an SQL statement for querying the row created in Review Question 10B.14.

10B.16 Explain the problems inherent in using sequences for surrogate key columns.

10B.17 Show SQL statements for creating a relationship between table T2 and table T3. Assume that T3 has a foreign key column named FK1 that relates to T2 and that deletions in T2 should force deletions in T3.

10B.18 Answer Review Question 10B.17, but do not force deletions.

10B.19 Explain how to use the TO_DATE function.

10B.20 Explain how you would use SQL Developer to create an index on the salary field of an employee table. What sorts of queries will benefit from this, and how?

If you have not already installed Oracle Database XE (or do not otherwise have a version of Oracle Database available to you), you need to install a version of it and Oracle SQL Developer at this point.

Review Questions 10B.21–10B.34 are based on a database named MEDIA that is used to record data about photographs that are stored in the database.

10B.21 Create a database named *MEDIA* in Oracle Database.

10B.22 In the SQL Developer folder structure in your *Documents* folder, create a folder named *DBP-e14-Media-Database*. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining Review Questions in this section.

10B.23 Create a connection named Media-Database in SQL Developer, and use it to connect to the MEDIA database. Open a new tabbed SQL Worksheet window, and save it as *MEDIA-CH10B-RQ-Solutions.sql* in the *DBP-e14-Media-Database* folder. Use this script to record and save the SQL statements that you are asked to create in the remaining Review Questions in this section.

10B.24 Write an SQL CREATE TABLE statement to create a table named PICTURE using the column characteristics as shown in Figure 10B-95. Run the SQL statement to create the PICTURE table in the MEDIA database.

10B.25 Write an SQL CREATE TABLE statement to create the table SLIDE_SHOW using the column characteristics, as shown in Figure 10B-96. Run the SQL statement to create the SLIDE_SHOW table in the MEDIA database.

10B.26 Write an SQL CREATE TABLE statement to create the table SLIDE_SHOW_PICTURE_INT using the column characteristics, as shown in Figure 10B-97. SLIDE_SHOW_PICTURE_INT is an intersection table between PICTURE and SLIDE_SHOW, so create appropriate relationships between PICTURE and SLIDE_SHOW_PICTURE_INT and between SLIDE_SHOW and SLIDE_SHOW_PICTURE_INT. Set the referential integrity properties to disallow any deletion of a SLIDE_SHOW row that has any SLIDE_SHOW_PICTURE_INT rows related to it. Set the referential integrity properties to cascade deletions in the intersection table when a PICTURE is deleted.

10B.27 Write SQL INSERT statements to populate the PICTURE table using the data shown in Figure 10B-98. Run the SQL statements to populate the PICTURE table.

FIGURE 10B-95
Column Characteristics for the MEDIA Database PICTURE Table

Column Name	Type	Key	Required	Remarks
PictureID	Integer	Primary Key	Yes	Surrogate Key: Initial value=1 Increment=1
PictureName	Character (35)	No	Yes	
PictureDescription	Varchar (255)	No	No	Default “None”
DateTaken	Date	No	Yes	
PictureFileName	Varchar (45)	No	Yes	

FIGURE 10B-96
Column Characteristics for the MEDIA Database SLIDE_SHOW Table

Column Name	Type	Key	Required	Remarks
ShowID	Integer	Primary Key	Yes	Surrogate Key: Initial value = 1000 Increment = 1
ShowName	Character (35)	No	Yes	
ShowDescription	Varchar (255)	No	No	Default "None"
Purpose	Character (15)	No	Yes	Data value must be one of the following: Home Office Family Recreation Sports Pets

FIGURE 10B-97
Column Characteristics for the MEDIA Database SLIDE_SHOW_PICTURE_INT Table

Column Name	Type	Key	Required	Remarks
ShowID	Integer	Primary Key, Foreign Key	Yes	REF: SLIDE_SHOW
PictureID	Integer	Primary Key, Foreign Key	Yes	REF: PICTURE

10B.28 Write SQL INSERT statements to populate the SLIDE_SHOW table using the data shown in Figure 10B-99. Run the SQL statements to populate the SLIDE_SHOW table.

10B.29 Write SQL INSERT statements to populate the SLIDE_SHOW_PICTURE_INT table using the data shown in Figure 10B-100. Run the SQL statements to populate the SLIDE_SHOW_PICTURE_INT table.

10B.30 Write an SQL statement to create a view named *PopularShowsView* that has SLIDE_SHOW.ShowName and PICTURE.PictureName for all slide shows that have a Purpose of either 'Home' or 'Pets'. Execute this statement to create the view in the MEDIA database.

10B.31 Run an SQL SELECT query to demonstrate that the view PopularShowsView was constructed correctly.

10B.32 Use the SQL Developer GUI tools to determine that the PopularShowsView view was constructed correctly. Modify this view to include PICTURE.PictureDescription and PICTURE.PictureFileName. *Hint:* Clicking on the pencil icon in the Columns tab of the main view tab will allow you to edit the SQL defining the view.

FIGURE 10B-98
Sample Data for the MEDIA Database PICTURE Table

PictureID	PictureName	PictureDescription	Date Taken	PictureFileName
1	SpotAndBall	My dog Spot chasing a ball	2015-09-07	spot00001.jpg
2	SpotAndCat	My dog Spot chasing a cat	2015-09-08	spot00002.jpg
3	SpotAndCar	My dog Spot chasing a car	2015-10-11	spot00003.jpg
4	SpotAndMailman	My dog Spot chasing a mailman - BAD DOG!	2015-11-22	spot00004.jpg
5	TheJudgeAndI	I explain that Spot is really a good dog, and did not mean to chase the mailman	2015-12-13	me00001.jpg

FIGURE 10B-99
Sample Data for the MEDIA Database SLIDE_SHOW Table

ShowID	ShowName	ShowDescription	Purpose
1000	My Dog Spot	My dog Spot likes to chase things	Pets
1001	My Day In Court	I explain that Spot is really a good dog	Home

10B.33 Can the SQL DELETE statement be used with the *PopularShowsView* view? Why or why not?

10B.34 Under what circumstances can the *PopularShowsView* view be used for inserts and modifications?

Review Questions 10B.35–10B.38 are based on the VRG database discussed in this chapter.

10B.35 For the View Ridge Gallery database discussed in this chapter, construct a view that contains a customer's LastName, FirstName, City, and State. Name your view *CustomerBasicView*.

10B.36 For the View Ridge Gallery database, construct a view that has the full customer name and full artist name for all art that the customer has purchased.

10B.37 For the View Ridge Gallery database, construct a view that has full customer name and full artist name for all artists in which the customer is interested. Explain the difference between this view and the view in Review Question 10B.36.

10B.38 Can you combine the views in Review Questions 10B.36 and 10B.37 into one view? Why or why not?

10B.39 How can you update an SQL view using Oracle Database?

10B.40 In PL/SQL, what is the purpose of the RETURN keyword?

10B.41 What must be done to be able to see the output generated by the Oracle Database DBMS_OUTPUT package? What limits exist on such output?

10B.42 Explain how the PL/SQL statement FOR variable IN cursor name works.

10B.43 Where in SQL Developer will you see error messages when compiling stored procedures and triggers?

10B.44 What is the syntax of the BEGIN TRANSACTION statement in PL/SQL? How is a transaction started?

10B.45 In the stored procedure in Figure 10B-82, how are the values of the variables varTID and varAID used if there are no suitable TRANS rows in the database? How are they used if there is just one suitable TRANS row in the database?

10B.46 Explain the purpose of BEFORE, AFTER, and INSTEAD OF triggers.

FIGURE 10B-100
Sample Data for the MEDIA Database SLIDE_SHOW_PICTURE_INT Table

ShowID	PictureID
1000	1
1000	2
1000	3
1000	4
1001	4
1001	5

10B.47 When an update is in progress, how can the trigger code obtain the value of a column, say C1, before the update began? How can the trigger code obtain the value that the column is being set to?

10B.48 Explain why INSTEAD OF triggers are needed for join views.

10B.49 Explain a limitation on the use of AFTER triggers.

10B.50 What three levels of transaction isolation are supported by Oracle Database?

10B.51 Explain how Oracle Database uses the system change number (SCN) to read data that are current at a particular point in time.

10B.52 Under what circumstances does Oracle Database read dirty data?

10B.53 Explain how conflicting locks are handled by Oracle Database when a transaction is operating in read-committed isolation mode.

10B.54 Show the SQL statement necessary to set the transaction isolation level to serializable for an entire session.

10B.55 What happens when a transaction in serializable mode tries to update data that have been updated by a different transaction? Assume that the SCN is less than the transaction's SCN. Assume the SCN is greater than the transaction's SCN.

10B.56 Describe three circumstances under which a transaction could receive the "Cannot serialize" exception.

10B.57 Explain how Oracle Database processes the read-only transaction isolation level.

10B.58 What three types of files are important for Oracle Database backup and recovery processing?

10B.59 What is the difference between the OnLine ReDo logs and the OffLine or Archive ReDo logs? How is each type used?

10B.60 What does multiplexing mean in the context of Oracle Database recovery?

10B.61 Explain how Oracle Database recovers from application failure.

10B.62 What is an instance failure, and how does Oracle Database recover from it?

10B.63 What is a media failure, and how does Oracle Database recover from it?

Project Questions

In the Chapter 7 Review Questions, we introduced the Wedgewood Pacific Corporation (WPC) and developed the WPC database. Two of the tables that are used in the WPC database are:

DEPARTMENT (DepartmentName, BudgetCode, OfficeNumber, Phone)

EMPLOYEEE (EmployeeNumber, FirstName, LastName, *Department*, Phone, Email)

Assume that the relationship between these tables is M-M, and use the tables as the basis for your answers to Review Questions 10B.64–10B.68.

10B.64 In the Oracle SQL Developer folder structure in your My Documents folder, create a folder named *DBP-e14-WPC-CH10B-PQ-Database*. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining questions in this section.

10B.65 Using the examples in this chapter as templates:

- For *Oracle Database 12c*: Use the Oracle Enterprise Manager to create a tablespace named WPCCH10BPQ, a user named WPC_CH10B_PQ_USER with a password of WPC_CH10B_PQ_USER+password, and a role named WPCCH10BPQ_DEV that has the CREATE VIEW system privilege. AssignWPC_CH10B_PQ_USER the CONNECT, RESOURCE, and WPCCH10BPQ_DEV roles.
- For *Oracle Database XE*: Use the Oracle Database XE11.2 Web utility to create a WPC_CH10B_PQ workspace with user accounts WPC_CH10B_USER.

10B.66 Using the information about the WPC database in the Chapter 7 Review Questions and the referenced figures in Chapter 1, create the EMPLOYEE and DEPARTMENT tables and the relationship between these tables.

10B.67 Using the information about the WPC database in the Chapter 7 Review Questions and the referenced figures in Chapter 1, populate the EMPLOYEE and DEPARTMENT tables.

10B.68 Code an Oracle trigger to enforce the constraint that an employee can never change his or her department.

10B.69 Code an Oracle trigger to allow the deletion of a department if it has only one employee. Assign the last employee to the Human Resources department.

10B.70 Design a system of triggers to enforce the M-M relationship. Use Figure 10B-91 as an example, but assume that departments with only one employee can be deleted. Assign the last employee in a department to Human Resources.

Project Question 10B.71 and 10B.72 are based on the View Ridge Gallery database discussed in this chapter. If you have not already installed Oracle Database XE (or do not otherwise have a version of Oracle Database available to you), you need to install it and Oracle SQL Developer at this point.

10B.71 Write SQL statements to accomplish the following tasks and submit them to Oracle Database using Oracle SQL Developer.

A. In the Oracle SQL Developer folder structure in your My Documents folder, create a folder named *DBP-e14-VRG-CH10B-PQ-Database*. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining questions in this section.

B. Using the examples in this chapter as templates:

- For *Oracle Database 12c*: Use the Oracle Enterprise Manager to create a tablespace named VRGCH10BPQ, a user named VRG_CH10B_PQ_USER with a password of VRG_CH10B_PQ_USER+password, and a role named VRGCH10BPQ_DEV that has the CREATE VIEW system privilege. Assign VRG_CH10B_PQ_USER the CONNECT, RESOURCE, and VRGCH10BPQ_DEV roles.
- For *Oracle Database XE*: Use the Oracle Database XE11.2 Web utility to create a VRG_CH10B_PQ workspace with user accounts VRG_CH10B_USER.

C. Create the tables in Figure 10B-48, but do not create the NationalityValues constraint.

D. Populate your database with the data shown in Figure 10B-59.

E. Write a stored procedure to read the ARTIST table, and display the artist data using the DBMS_OUTPUT.PUT_LINE command. Specifically, your procedure should have one parameter (the artist last name) and should display all columns except ArtistID for every artist with that last name.

F. Write a stored procedure to read the ARTIST and WORK tables. Your procedure should display an artist, then display all the works for that artist. Accept the first and last names of the artist to display as input parameters.

G. Write a stored procedure to update customer phone data. Assume that your stored procedure receives LastName, FirstName, priorAreaCode, newAreaCode, priorPhoneNumber, and newPhoneNumber. Your procedure should first ensure that there is only one customer with the values of (LastName, FirstName, priorAreaCode, priorPhoneNumber). If not, produce an error message and quit. Otherwise, update the customer data with the new phone number data.

H. Create a table named ALLOWED_NATIONALITY with one column called Nation. Place the values 'Canadian', 'English', 'French', 'German', 'Mexican', 'Russian', 'Spanish', and 'United States' into the table. Write a trigger that will check to determine whether a new or updated value of Nationality resides in this table. If not, disallow the insert or update, and write an error message using DBMS_OUTPUT.PUT_LINE. Use SQL Developer to demonstrate that your trigger works.

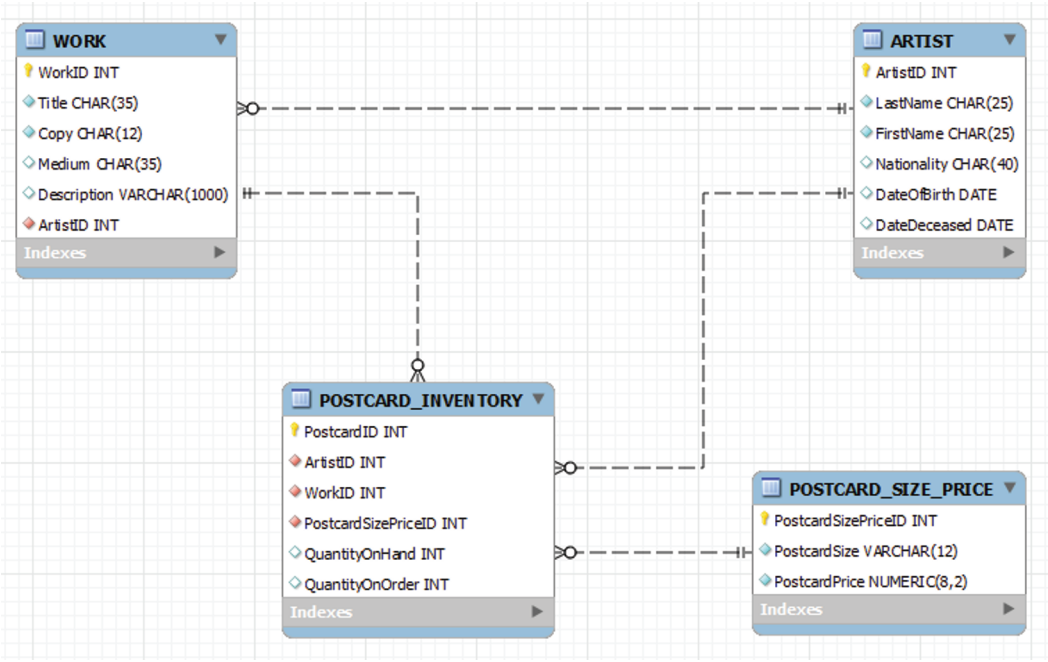
10B.72 Write SQL statements to accomplish the following tasks, and submit them to Oracle Database via SQL Developer. Save your work in an SQL script named *VRG-CH10B-PQ-10B-70.sql*. This Project Question shows the steps necessary to integrate the *POSTCARDS_TEMP* table data into the VRG database. A database diagram showing how the VRG database will appear after these steps are completed is shown in Figure 10B-101.

A. If you haven't done so, work through Project Question 10B.69 to create the Oracle Database database named *VRG-CH10B-PQ* as described in that Project Question.

B. Use the steps described in this chapter to:

- Create a Microsoft Excel 2013 workbook containing the POSTCARDS and POSTCARDSwithID worksheets shown in Figure 10B-62.
- Import the data in the POSTCARDSwithID worksheet into a table in the VRG database name POSTCARDS_TEMP. *Hint:* Be careful with Char and Varchar data types. For example, on importing, ensure that the types of WORK.Title and POSTCARDS_TEMP.WorkTitle are the same.
- Create the *GetLastNameCommaSeparated* user-defined function shown in Figure 10B-76.

FIGURE 10B-101
Partial Database Design for the Revised VRG Database



- Alter the POSTCARDS_TEMP table to include the ArtistLastName and ArtistID columns as discussed in the text and as shown in Figure 10B-78.
- Populate the POSTCARDS_TEMP table ArtistLastName and ArtistID columns as discussed in the text and as shown in Figure 10B-79.

C. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order and with the names separated by a comma and one space. Write an SQL SELECT statement using the POSTCARDS_TEMP table to test your function. *Hint:* You may interpret "first name" as including both first and middle names, if present, for the purposes of this question.

D. Alter the POSTCARDS_TEMP table to include an ArtistFirstName column (Integer data, allow NULL values). Use the *GetFirstNameCommaSeparated* function that you created in part C to populate this column.

E. Alter the POSTCARDS_TEMP table to include a WorkID column (Char(25) data, allow NULL values). By using and comparing the data in the POSTCARDS_TEMP.WorkTitle and the WORK.Title columns, populate this column. *Hint:* If you were not careful about data types during the import process, you may need to use the RTRIM function (see Chapter 2 page 73) to properly compare WORK.Title to POSTCARDS_TEMP.WorkTitle. Also, since multiple works share the same title, and we only want one of those to correspond to the postcard, the SQL SELECT TOP 1… syntax (see Chapter 2 page 62) would ensure that just one WorkID is returned from a query. Oracle, however, does not directly support that syntax. Instead, use Oracle's built-in "ROWNUM" variable. For example, "SELECT SSN FROM EMPLOYEE WHERE LastName = 'Kroenke' AND ROWNUM = 1;" will retrieve the first employee it finds with the last name 'Kroenke'.

F. Create a new table named POSTCARD_SIZE_PRICE. Use the column characteristics shown in Figure 10B-101, where PostcardSizePriceID is a surrogate key starting at 1 and incrementing by 1.

G. Populate the POSTCARD_SIZE_PRICE table using the data stored in the POSTCARDS_TEMP table. *Hint:* You should insert distinct data into the table, and your final table will have only 3 records. Also note that using "nextVal" with a sequence will not work with the DISTINCT clause, so if you want to do this with one INSERT INTO…SELECT statement, use a subquery as shown in Chapter 2 on page 78.

H. Alter the POSTCARDS_TEMP table to include a PostcardSizePriceID column (Integer data, allow NULL values). By using and comparing the data in the POSTCARDS_TEMP.PostCardSize and the POSTCARDS_SIZE_PRICE .PostCardSize columns, populate this column.

I. Create a new table named POSTCARD_INVENTORY. Use the column characteristics shown in Figure 10B-101, where PostcardID is a surrogate key starting at 1 and incrementing by 1.

J. Populate the POSTCARD_INVENTORY table using the data stored in the POSTCARDS_TEMP table. *Hint:* You will have 1 record in this table for every record in the POSTCARDS_TEMP table, and your final table will have only 26 records.

K. We have completed our modifications of the VRG database, and we are done with the temporary POSTCARDS_TEMP table. We *could* delete if we wanted to, but we will *keep the POSTCARDS_TEMP table* in the database.

Case Questions

Marcia's Dry Cleaning Case Questions

Marcia Wilson owns and operates *Marcia's Dry Cleaning*, which is an upscale dry cleaner in a well-to-do suburban neighborhood. Marcia makes her business stand out from the competition by providing superior customer service. She wants to keep track of each of her customers and their orders. Ultimately, she wants to notify them that their clothes are ready via email. Suppose that you have designed a database for Marcia's Dry Cleaning that has the following tables:

CUSTOMER (CustomerID, FirstName, LastName, Phone, EmailAddress)

INVOICE (InvoiceNumber, *CustomerID*, DateIn, DateOut, Subtotal, Tax, TotalAmount)

INVOICE_ITEM (*InvoiceNumber*, ItemNumber, *ServiceID*, Quantity, UnitPrice, ExtendedPrice)

SERVICE (ServiceID, ServiceDescription, UnitPrice)

The referential integrity constraints are:

CustomerID in INVOICE must exist in CustomerID in CUSTOMER

InvoiceNumber in INVOICE_ITEM must exist in InvoiceNumber in INVOICE

ServiceID in INVOICE_ITEM must exist in ServiceID in SERVICE

Assume that CustomerID of CUSTOMER and InvoiceNumber of INVOICE are surrogate keys with values as follows:

CustomerID	**Start at 100**	**Increment by 1**
InvoiceNumber	**Start at 2015001**	**Increment by 1**

Further, assume that ServiceID is a surrogate key, but not one that automatically increments—the values of ServiceID are assigned by Marcia's Dry Cleaning management when new services are added at Marcia's Dry Cleaning.

A. Specify NULL/NOT NULL constraints for each table column.

B. Specify alternate keys, if any.

C. State relationships as implied by foreign keys, and specify the maximum and minimum cardinality of each relationship. Justify your choices.

D. Explain how you will enforce the minimum cardinalities in your answer to part C. Use referential integrity actions for required parents, if any. Use Figure 6-29(b) as a boilerplate for required children, if any.

E. In the SQL Developer folder structure in your *My Documents* folder, create a folder named *DBP-e14-MDC-Database* in the *Projects* folder. Use this folder to save and store *.sql scripts containing the SQL statements that you are asked to create in the remaining questions in this section.

F. Using the examples in this chapter as a template:

- For *Oracle Database 12c*: Use the Oracle Enterprise Manager to create a tablespace named MDCCH10B, a user named MDC_CH10B_USER with a password of MDC_CH10B_USER+password, and a role named MDCCH10B_DEV that has the CREATE VIEW system privilege. Assign MDC_CH10B_USER the CONNECT, RESOURCE, and MDCCH10B_DEV roles.

- For *Oracle Database XE*: Use the Oracle Database XE11.2 Web utility to create a MDC_CH10B_CQ workspace with user accounts MDC_CH10B_USER.

Using the MDC database, create an SQL script named *MDC-Create-Tables.sql* to answer parts G and H. Your answer to part H should be in the form of a comment in the SQL script.

G. Write CREATE TABLE statements for each of the tables using your answers to parts A–D, as necessary. Set the first value of CustomerID to 100 and increment it by 1. Use FOREIGN KEY constraints to create appropriate referential integrity constraints. Set UPDATE and DELETE behavior in accordance with your referential integrity action design. Set the default value of Quantity to 1. Write a constraint that SERVICE.UnitPrice be between 1.50 and 10.00.

H. Explain how you would enforce the data constraint that INVOICE_ITEM.UnitPrice be equal to SERVICE.UnitPrice, where INVOICE_ITEM.ServiceID = SERVICE.ServiceID.

Using the MDC database, create an SQL script named *MDC-Insert-Data.sql* to answer part I.

I. Write INSERT statements to insert the data shown in Figures 10B-102, 10B-103, 10B-104, and 10B-105.

FIGURE 10B-102
Sample Data for the MDC Database CUSTOMER Table

CustomerID	FirstName	LastName	Phone	Email
100	Nikki	Kaccaton	723-543-1233	Nikki.Kaccaton@somewhere.com
101	Brenda	Catnazaro	723-543-2344	Brenda.Catnazaro@somewhere.com
102	Bruce	LeCat	723-543-3455	Bruce.LeCat@somewhere.com
103	Betsy	Miller	723-654-3211	Betsy.Miller@somewhere.com
104	George	Miller	723-654-4322	George.Miller@somewhere.com
105	Kathy	Miller	723-514-9877	Kathy.Miller@somewhere.com
106	Betsy	Miller	723-514-8766	Betsy.Miller@elsewhere.com

FIGURE 10B-103
Sample Data for the MDC Database SERVICE Table

ServiceID	ServiceDescription	UnitPrice
10	Mens Shirt	$1.50
11	Dress Shirt	$2.50
15	Women's Shirt	$1.50
16	Blouse	$3.50
20	Slacks— Men's	$5.00
25	Slacks—Women's	$6.00
30	Skirt	$5.00
31	Dress Skirt	$6.00
40	Suit– Men's	$9.00
45	Suit– Women's	$8.50
50	Tuxedo	$10.00
60	Formal Gown	$10.00

FIGURE 10B-104
Sample Data for the MDC Database INVOICE Table

InvoiceNumber	CustomerID	DateIn	DateOut	SubTotal	Tax	TotalAmount
2015001	100	04-Oct-15	06-Oct-15	$158.50	$12.52	$171.02
2015002	101	04-Oct-15	06-Oct-15	$25.00	$1.98	$26.98
2015003	100	06-Oct-15	08-Oct-15	$49.00	$3.87	$52.87
2015004	103	06-Oct-15	08-Oct-15	$17.50	$1.38	$18.88
2015005	105	07-Oct-15	11-Oct-15	$12.00	$0.95	$12.95
2015006	102	11-Oct-15	13-Oct-15	$152.50	$12.05	$164.55
2015007	102	11-Oct-15	13-Oct-15	$7.00	$0.55	$7.55
2015008	106	12-Oct-15	14-Oct-15	$140.50	$11.10	$151.60
2015009	104	12-Oct-15	14-Oct-15	$27.00	$2.13	$29.13

FIGURE 10B-105
Sample Data for the MDC Database INVOICE_ITEM Table

InvoiceNumber	ItemNumber	ServiceID	Quantity	UnitPrice	ExtendedPrice
2015001	1	16	2	$3.50	$7.00
2015001	2	11	5	$2.50	$12.50
2015001	3	50	2	$10.00	$20.00
2015001	4	20	10	$5.00	$50.00
2015001	5	25	10	$6.00	$60.00
2015001	6	40	1	$9.00	$9.00
2015002	1	11	10	$2.50	$25.00
2015003	1	20	5	$5.00	$25.00
2015003	2	25	4	$6.00	$24.00
2015004	1	11	7	$2.50	$17.50
2015005	1	16	2	$3.50	$7.00
2015005	2	11	2	$2.50	$5.00
2015006	1	16	5	$3.50	$17.50
2015006	2	11	10	$2.50	$25.00
2015006	3	20	10	$5.00	$50.00
2015006	4	25	10	$6.00	$60.00
2015007	1	16	2	$3.50	$7.00
2015008	1	16	3	$3.50	$10.50
2015008	2	11	12	$2.50	$30.00
2015008	3	20	8	$5.00	$40.00
2015008	4	25	10	$6.00	$60.00
2015009	1	40	3	$9.00	$27.00

Using the MDC database, create an SQL script named *MDC-DML-CH10B.sql* to answer parts J and K.

J. Write an UPDATE statement to change values of SERVICE.Description from Mens Shirt to Mens' Shirts.

K. Write a DELETE statement(s) to delete an INVOICE and all of the items on that INVOICE.

Using the MDC database, create an SQL script named *MDC-Create-Views-and-Functions.sql* to answer parts L through T. Your answer to part P should be in the form of a comment in the SQL script.

L. Create a view called OrderSummaryView that contains INVOICE.InvoiceNumber, INVOICE.DateIn, INVOICE.DateOut, INVOICE_ITEM.ItemNumber, INVOICE_ITEM.ServiceID, and INVOICE_ITEM.ExtendedPrice.

M. Create a view called CustomerOrderSummaryView that contains INVOICE.InvoiceNumber, CUSTOMER.FirstName, CUSTOMER.LastName, CUSTOMER.Phone, INVOICE.DateIn, INVOICE.DateOut, INVOICE.SubTotal, INVOICE_ITEM.ItemNumber, INVOICE_ITEM.ServiceID, and INVOICE_ITEM.ExtendedPrice.

N. Create a view called CustomerOrderHistoryView that (1) includes all columns of CustomerOrderSummaryView except INVOICE_ITEM.ItemNumber and INVOICE_ITEM.Service; (2) groups orders by CUSTOMER.LastName, CUSTOMER.FirstName, and INVOICE.InvoiceNumber, in that order; and (3) sums and averages INVOICE_ITEM.ExtendedPrice for each order for each customer. *Hint:* In (2), note that the group by will also have to include Phone, DateIn, etc., since the selected columns must be a subset of the grouping columns.

O. Create a view called CustomerOrderCheckView that uses CustomerOrderHistoryView and that shows any customers for whom the sum of INVOICE_ITEM.ExtendedPrice is not equal to INVOICE.SubTotal.

P. Explain, in general terms, how you will use triggers to enforce minimum cardinality actions as required by your design. You need not write the triggers, just specify which triggers you need and describe, in general terms, their logic.

Q. Create and test a user-defined function named *LastNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *LastName, FirstName* (including the comma and space).

R. Create and test a view called CustomerOrderSummaryView that contains the customer name concatenated and formatted as *LastName, FirstName* in a field named CustomerName, INVOICE.InvoiceNumber, INVOICE.DateIn, INVOICE.DateOut, and INVOICE.TotalAmount.

S. Create and test a user-defined function named *FirstNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *FirstName LastName* (including the space).

T. Create and test a view called CustomerDataView that contains the customer name concatenated and formatted as *FirstName LastName* in a field named CustomerName, Phone, and Email.

Using the MDC database, create an SQL script named *MDC-Create-Triggers.sql* to answer parts U and V.

U. Assume that the relationship between INVOICE and INVOICE_ITEM is M-M. Design triggers to enforce this relationship. Use Figure 10B-96 and the discussion of that figure as an example, but assume that Marcia does allow INVOICEs and their related INVOICE_ITEM rows to be deleted. Use the deletion strategy shown in Figures 7-28 and 7-29 for this case.

V. Write and test the triggers you designed in part U.

Marcia's Dry Cleaning tracks which employees have worked on specific dry cleaning jobs. This is somewhat complicated by the fact that more than one employee may have helped a customer with a particular order. So far, the company has kept their insurance records in a Microsoft Excel 2013 worksheet, as shown in Figure 10B-106. They have decided to integrate this data into the MDC database. The modifications to the MDC database needed to accomplish this are shown in Figure 10B-107 (as a MySQL Workbench EER diagram). Using the MDC database, create an SQL script named *MDC-Import-Excel-Data.sql* to answer parts W through AI.

W. Duplicate the EMPLOYEE worksheet Figure 10B-106 a worksheet (or spreadsheet) in a Mircosoft Excel 2013 (or another tool such as Apache OpenOffice Calc).

X. Import the data in the EMPLOYEE worksheet into a table in the MDC database named EMPOYEE_TEMP.

Y. Create the *GetLastNameCommaSeparated* user-defined function shown is Figure 10B-76.

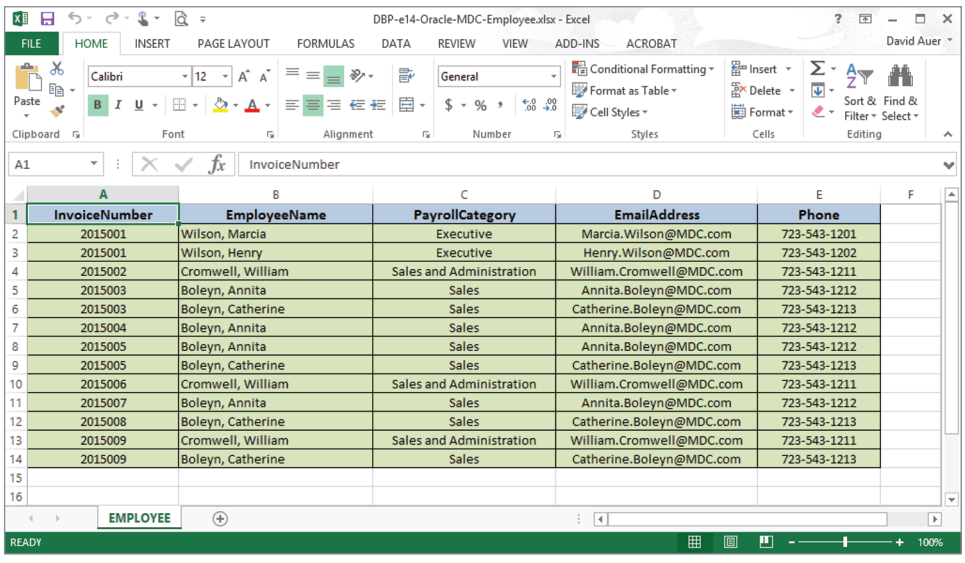
Z. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order, with the names separated by a comma and one space.

AA. Alter the EMPLOYEE_TEMP table to include EmployeeLastName and EmployeeFirstName columns (Char(25), allow NULL values).

AB. Use the *GetLastNameCommaSeparated* user-defined function you created in step Y to populate the EmployeeLastName column.

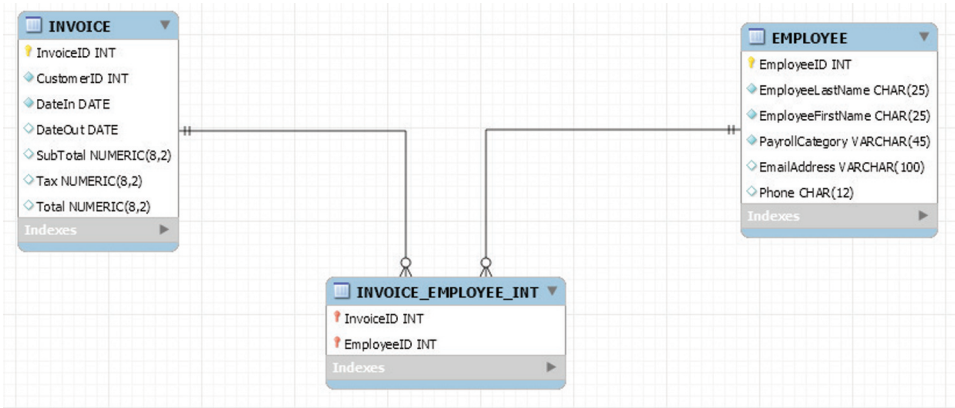
AC. Use the *GetFirstNameCommaSeparated* user-defined function you created in step Z to populate the EmployeeFirstName column.

FIGURE 10B-106
The Marcia's Dry Cleaning Employee Worksheet



	A	B	C	D	E	F
1	InvoiceNumber	EmployeeName	PayrollCategory	EmailAddress	Phone	
2	2015001	Wilson, Marcia	Executive	Marcia.Wilson@MDC.com	723-543-1201	
3	2015001	Wilson, Henry	Executive	Henry.Wilson@MDC.com	723-543-1202	
4	2015002	Cromwell, William	Sales and Administration	William.Cromwell@MDC.com	723-543-1211	
5	2015003	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
6	2015003	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
7	2015004	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
8	2015005	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
9	2015005	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
10	2015006	Cromwell, William	Sales and Administration	William.Cromwell@MDC.com	723-543-1211	
11	2015007	Boleyn, Annita	Sales	Annita.Boleyn@MDC.com	723-543-1212	
12	2015008	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
13	2015009	Cromwell, William	Sales and Administration	William.Cromwell@MDC.com	723-543-1211	
14	2015009	Boleyn, Catherine	Sales	Catherine.Boleyn@MDC.com	723-543-1213	
15						
16						

FIGURE 10B-107
Partial Database Design for the Modified MDC Database



AD. Create a new table named EMPLOYEE, as shown in Figure 10B-107. Use the column characteristics shown in Figure 10B-107, where EmployeeID is a surrogate key starting at 1 and incrementing by 1.

AE. Populate the EMPLOYEE table using the data stored in the EMPLOYEE_TEMP table. *Hint:* You should insert distinct data into the table, and your final table will have only 5 records.

AF. Alter the EMPLOYEE_TEMP to include an EmployeeID column (Integer data, allow nulls). By using and comparing the EMPLOYEE_TEMP.EmployeeLastName and EMPLOYEE_TEMP.EmployeeFirstName columns with the EMPLOYEE .EmployeeLastName and EMPLOYEE.EmployeeFirstName columns, populate this column. *Hint:* Assume for this question that no two employees have the same first and last names.

AG. Create a new table named INVOICE_EMPLOYEE_INT, as shown in Figure 10B-107. Use the column characteristics shown in Figure 10B-107.

AH. Populate the INVOICE_EMPLOYEE_INT table using the data stored in the EMPLOYEE_TEMP table. *Hint:* You will have one record in the INVOICE_EMPLOYEE table for every record in the EMPLOYEE_TEMP table, and your final table will have 13 records.

AI. We have completed the modifications of the MDC database, and are done with the temporary EMPLOYEE_TEMP table. We *could* delete it if we wanted to, but we will *keep the EMPLOYEE_TEMP table* in the database.

The Queen Anne Curiosity Shop

If you have not completed the discussion of the Queen Anne Curiosity Shop database at the end of Chapter 7, work through the Chapter 7 QACS Project Questions now. Use the QACS database that you created in the Chapter 7 QACS Project Questions as the basis for your answers to the following questions:

A. Using the examples in this chapter as a template:

- For *Oracle Database 12c*: Use the Oracle Enterprise Manager to create a tablespace named QACSCH10B, a user named QACS_CH10B_USER with a password of QACS_CH10B_USER+password, and a role named QACSCH10B_DEV that has the CREATE VIEW system privilege. Assign QACS_CH10B_USER the CONNECT, RESOURCE, and QACSCH10B_DEV roles.
- For *Oracle Database XE*: Use the Oracle Database XE11.2 Web utility to create a QACS_CH10B workspace with user account QACS_CH10B_USER.

Using the QACS database, create an SQL script named *QACS-Create-Views-and-Functions.sql* to answer parts B through F.

B. Create and test a user-defined function named *LastNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *LastName, FirstName* (including the comma and space).

C. Create and test a view called CustomerSaleSummaryView that contains the customer name concatenated and formatted as *LastName, FirstName* in a field named CustomerName, SALE.SaleID, SALE.Date, and SALE.Total.

D. Create and test a user-defined function named *FirstNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *FirstName LastName* (including the space).

E. Create and test a user-defined function named *CityStateZIP* that combines three parameters named *City, State*, and *ZIP* into a concatenated address field formatted *City, State ZIP* (including the comma and the spaces).

F. Create and test a view called CustomerMailingAddressView that contains the customer name concatenated and formatted as *FirstName LastName* in

a field named CustomerName, the customer's street address in a field named CustomerStreetAddress, and the customer's City, State, ZIP concatenated and formatted as *City, State ZIP* in a field named CustomerCityStateZIP.

Using the QACS database, create an SQL script named *QACS-Create-Triggers.sql* to answer parts G and H.

G. Assume that the relationship between SALE and SALE_ITEM is M-M. Design triggers to enforce this relationship. Use Figure 10B-91 and the discussion of that figure as an example, but assume that the Queen Anne Curiosity Shop does allow INVOICEs and their related INVOICE_ITEM rows to be deleted. Use the deletion strategy shown in Figures 7-28 and 7-29 for this case.

H. Write and test the triggers you designed in part G.

The Queen Anne Curiosity Shop payroll is paid twice monthly, once on the 10th of the month and once on the 25th of the month. Pay is determined by the type of job (the payroll category) and the number of hours worked (rounded to a whole number). Of course, the QACS owners keep detailed payroll records. So far, they have kept their records for these items in a Microsoft Excel worksheet, as shown in Figure 10B-108. They have decided to integrate this data into the QACS database. The modifications needed to accomplish this are shown in Figure 10B-109 (in a MySQL Workbench EER diagram). Using the QACS database, create an SQL script named *QACS-Import-Excel-Data.sql* to answer parts I through T.

I. Duplicate the PAYROLL worksheet in Figure 10B-108 in a worksheet (or spreadsheet) in an appropriate tool (such as Microsoft Excel or Apache OpenOffice Calc).

J. Import the data in the PAYROLL worksheet into a table in the MI database named PAYROLL_TEMP.

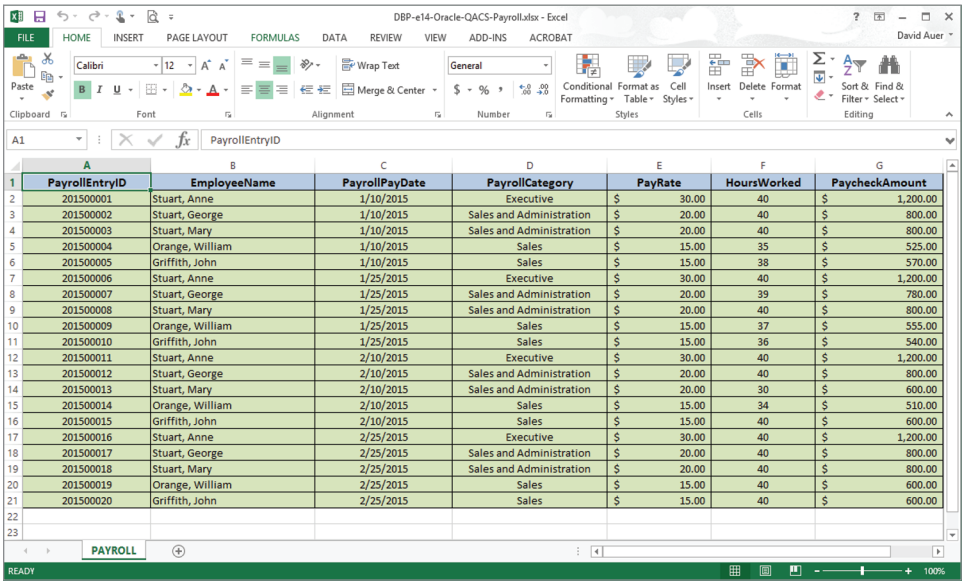
K. Create the *GetLastNameCommaSeparated* user-defined function shown is Figure 10B-76.

L. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order, with the names separated by a comma and one space.

M. Alter the PAYROLL_TEMP table to include EmployeeLastName and EmployeeFirstName columns (Char(25), allow NULL values).

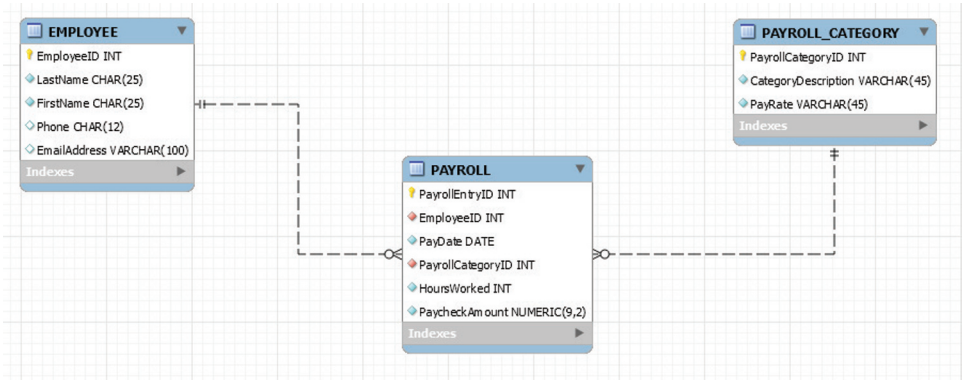
N. Use the *GetLastNameCommaSeparated* user-defined function you created in step K to populate the EmployeeLastName column.

FIGURE 10B-108
The Queen Anne Curiosity Shop Payroll Worksheet



PayrollEntryID	EmployeeName	PayrollPayDate	PayrollCategory	PayRate	HoursWorked	PaycheckAmount
201500001	Stuart, Anne	1/10/2015	Executive	$ 30.00	40	$ 1,200.00
201500002	Stuart, George	1/10/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500003	Stuart, Mary	1/10/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500004	Orange, William	1/10/2015	Sales	$ 15.00	35	$ 525.00
201500005	Griffith, John	1/10/2015	Sales	$ 15.00	38	$ 570.00
201500006	Stuart, Anne	1/25/2015	Executive	$ 30.00	40	$ 1,200.00
201500007	Stuart, George	1/25/2015	Sales and Administration	$ 20.00	39	$ 780.00
201500008	Stuart, Mary	1/25/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500009	Orange, William	1/25/2015	Sales	$ 15.00	37	$ 555.00
201500010	Griffith, John	1/25/2015	Sales	$ 15.00	36	$ 540.00
201500011	Stuart, Anne	2/10/2015	Executive	$ 30.00	40	$ 1,200.00
201500012	Stuart, George	2/10/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500013	Stuart, Mary	2/10/2015	Sales and Administration	$ 20.00	30	$ 600.00
201500014	Orange, William	2/10/2015	Sales	$ 15.00	34	$ 510.00
201500015	Griffith, John	2/10/2015	Sales	$ 15.00	40	$ 600.00
201500016	Stuart, Anne	2/25/2015	Executive	$ 30.00	40	$ 1,200.00
201500017	Stuart, George	2/25/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500018	Stuart, Mary	2/25/2015	Sales and Administration	$ 20.00	40	$ 800.00
201500019	Orange, William	2/25/2015	Sales	$ 15.00	40	$ 600.00
201500020	Griffith, John	2/25/2015	Sales	$ 15.00	40	$ 600.00

FIGURE 10B-109
Partial Database Design for the Modified QACS Database



O. Use the *GetFirstNameCommaSeparated* user-defined function you created in step L to populate the EmployeeFirstName column.

P. Create a new table named PAYROLL_CATEGORY, as shown in Figure 10B-109. Use the column characteristics shown in Figure 10B-109, where PayrollCategoryID is a surrogate key starting at 1 and incrementing by 1.

Q. Populate the PAYROLL_CATEGORY table using the data stored in the PAYROLL_TEMP table. *Hint:* You should insert distinct data into the table, and your final table will have only 3 records.

R. Alter the PAYROLL_TEMP table to include an EmployeeID column (Integer data, allow nulls). By using and comparing the PAYROLL_TEMP.EmployeeLastName and PAYROLL_TEMP.EmployeeFirstName columns with the EMPLOYEE.LastName and EMPLOYEE.FirstName columns, populate this column. *Hint:* Assume for this question that no two employees have the same first and last names.

S. Alter the PAYROLL_TEMP table to include a PayrollCategoryID column (Integer data, allow nulls). By using and comparing the PAYROLL_TEMP.PayrollCategory column with the PAYROLL_DESCRIPTION.CategoryDescription column, populate this column.

T. Create a new table named PAYROLL, as shown in Figure 10B-109. Use the column characteristics shown in Figure 10B-109. Note that PayrollEntryID is a surrogate key, with initial value 20150001 and incrementing by 1.

U. Populate the PAYROLL table using the data stored in the PAYROLL_TEMP table. *Hint:* You will have one record in the PAYROLL table for every record in the PAYROLL_TEMP table, and your final table will have 20 records.

V. We have completed the modifications of the MI database, and are done with the temporary PAYROL_TEMP table. We *could* delete it if we wanted to, but we will *keep the PAYROLL_TEMP table* in the database.

Morgan Importing

If you have not completed the discussion of the Morgan Importing database at the end of Chapter 7 on pages 390–395, work through the Chapter 7 Morgan Importing Project Questions now. Use the MI database that you created in the Chapter 7 QACS Project Questions as the basis for your answers to the following questions:

Note: In the SHIPMENT_RECEIPT table, the ReceiptDate and ReceiptTime columns should be combined into one column called ReceiptDateTime. This is because Oracle Database does not support a separate TIME data type but stores the date with the time in one DATE datatype. You may also think that TIMESTAMP would be a good data type to use, but despite their name the TIMESTAMP variants still include the year, month, and day, so they are *not* just a time data type, either, and not a solution to this problem. With a DATE datatype you can display DATE and TIME in almost any format you desire. See *http://docs.oracle.com/cd/E11882_01/server.112/e41084/sql_elements004.htm* for a further discussion of this function.

A. Using the examples in this chapter as a template:

- For *Oracle Database 12c*: Use the Oracle Enterprise Manager to create a tablespace named MICH10B, a user named MI_CH10B_USER with a password of MI_CH10B_USER+password, and a role named MICH10B_DEV that has the CREATE VIEW system privilege. Assign MI_CH10B_USER the CONNECT, RESOURCE, and MICH10B_DEV roles.
- For *Oracle Database XE*: Use the Oracle Database XE11.2 Web utility to create a MI_CH10B workspace with user account MI_CH10B_USER.

Using the MI database, create an SQL script named *MI-Create-Views-and-Functions.sql* to answer parts B through E.

B. Create and test a user-defined function named *LastNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *LastName, FirstName* (including the comma and space).

C. Create and test a view called *PurchasingAgentSummaryView* that contains the employee name of any MI employees who purchase items for the company, concatenated and formatted as *LastName, FirstName* (including the comma and space) in a field named PurchasingAgentName, ITEM.ItemDescription, ITEM.PurchaseDate, STORE.StoreName, STORE.City, and STORE.Country.

D. Create and test a user-defined function named *FirstNameFirst* that combines two parameters named *FirstName* and *LastName* into a concatenated name field formatted *FirstName LastName* (including the space).

E. Create and test a view called ReceivingAgentSummaryView that contains the employee name of any MI employees who received items for the company, concatenated and formatted as *FirstName Name* in a field named ReceivingAgentName, SHIPMENT_RECEIPT.ReceiptNumber, SHIPMENT.ShipmentID, SHIPPER.ShipperName, SHIPMENT.EstimatedArrivalDate, and the date and time of the receipt. *Hint:* use the Oracle TO_CHAR function to extract the date and time from the ReceiptDateTime field).

Using the MI database, create an SQL script named *MI-Create-Triggers.sql* to answer parts F and G.

F. Assume that the relationship between SHIPMENT and SHIPMENT_ITEM is M-M. Design triggers to enforce this relationship. Use Figure 10B-91 and the discussion of that figure as an example, but assume that Morgan does allow SHIPMENTs and their related SHIPMENT_ITEM rows to be deleted. Use the deletion strategy shown in Figures 7-28 and 7-29 for this case.

G. Write and test the triggers you designed in part F.

Morgan Importing purchases marine insurance to protect the company from monetary loss during shipping. So far, the company has kept their insurance records in a Microsoft Excel 2013 worksheet, as shown in Figure 10B-110. They have decided to integrate this data into the MI database. The modifications to the MI database needed to accomplish this are shown in Figure 10B-111 (as a MySQL Workbench EER diagram). Using using the MI database, create an SQL script named *MI-Import-Excel-Data.sql* to answer parts H through T.

H. Duplicate the INSURANCE worksheet Figure 10B-110 in a worksheet (or spreadsheet) in an Mircosoft Excel 2013 (or another tool such as Apache OpenOffice Calc).

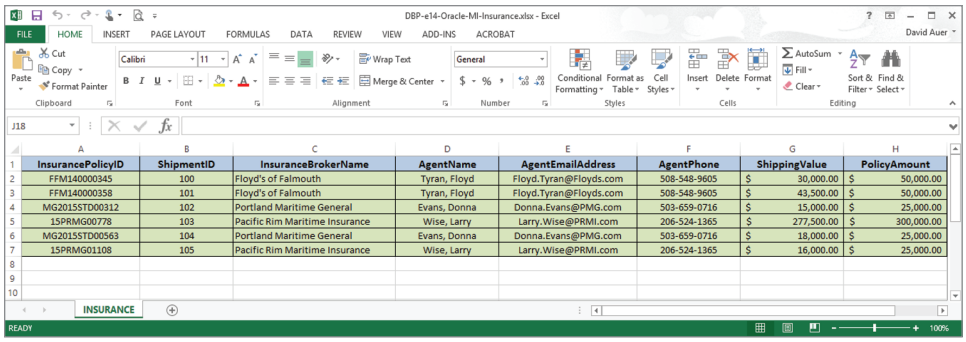
I. Import the data in the INSURANCE worksheet into a table in the MI database named INSURANCE_TEMP.

J. Create the *GetLastNameCommaSeparated* user-defined function shown is Figure 10B-76.

K. Create a user-defined function named *GetFirstNameCommaSeparated* that will return the first name from a combined name in last-name-first order, with the names separated by a comma and one space.

FIGURE 10B-110

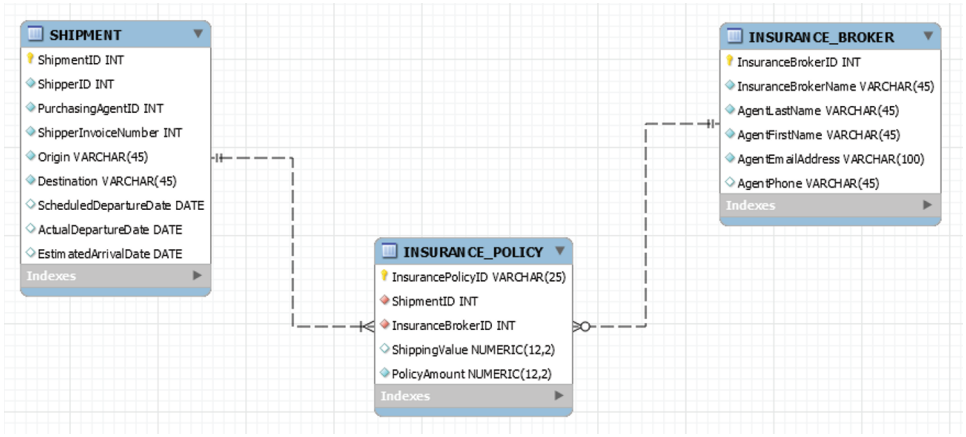
The Morgan Importing Maritime Insurance Worksheet



InsurancePolicyID	ShipmentID	InsuranceBrokerName	AgentName	AgentEmailAddress	AgentPhone	ShippingValue	PolicyAmount
FFM140000345	100	Floyd's of Falmouth	Tyran, Floyd	Floyd.Tyran@Floyds.com	508-548-9605	$ 30,000.00	$ 50,000.00
FFM140000358	101	Floyd's of Falmouth	Tyran, Floyd	Floyd.Tyran@Floyds.com	508-548-9605	$ 43,500.00	$ 50,000.00
MG2015STD00312	102	Portland Maritime General	Evans, Donna	Donna.Evans@PMG.com	503-659-0716	$ 15,000.00	$ 25,000.00
15PRMG00778	103	Pacific Rim Maritime Insurance	Wise, Larry	Larry.Wise@PRMI.com	206-524-1365	$ 277,500.00	$ 300,000.00
MG2015STD00563	104	Portland Maritime General	Evans, Donna	Donna.Evans@PMG.com	503-659-0716	$ 18,000.00	$ 25,000.00
15PRMG01108	105	Pacific Rim Maritime Insurance	Wise, Larry	Larry.Wise@PRMI.com	206-524-1365	$ 16,000.00	$ 25,000.00

FIGURE 10B-111

Partial Database Design for the Modified MI Database



L. Alter the INSURANCE_TEMP table to include AgentLastName and AgentFirstName columns (Varchar(45), allow NULL values).

M. Use the *GetLastNameCommaSeparated* user-defined function you created in step J to populate the AgentLastName column.

N. Use the *GetFirstNameCommaSeparated* user-defined function you created in step K to populate the AgentFirstName column.

O. Create a new table named INSURANCE_BROKER. Use the column characteristics shown in Figure 10B-111, where InsuranceBrokerID is a surrogate key starting at 1 and incrementing by 1.

P. Populate the INSURANCE_BROKER table using the data stored in the INSURANCE_TEMP table. *Hint:* You should insert distinct data into the table, and your final table will have only 3 records.

Q. Create a new table named INSURANCE_POLICY. Use the column characteristics shown in Figure 10B-111. Note that InsurancePolicyID is not a surrogate key, but rather uses a Varchar (25) character string.

R. Alter the INSURACE_TEMP table to include an InsuranceBrokerID column (Integer data, allow nulls). By using and comparing the INSURANCE_TEMP .InsuranceBrokerName and INSURANCE_BROKER.InsuranceBrokerName columns, populate this column. *Hint:* Assume for this question that no two insurance broker names are the same.

S. Populate the INSURANCE_POLICY table using the data stored in the INSURANCE_TEMP table. *Hint:* You will have one record in the INSURANCE_POLICY table for every record in the INSURANCE_TEMP table, and your final table will have 6 records.

T. We have completed the modifications of the MI database, and are done with the temporary INSURANCE_TEMP table. We *could* delete it if we wanted to, but we will keep the *INSURANCE_TEMP table* in the database.