CHAPTER 1

INTRODUCTION TO PL/SQL

LEARNING OBJECTIVES

After completing this chapter, you should be able to understand:

- PL/SQL and application programming
- Application models
- How to locate Oracle resources
- SQL and PL/SQL tools
- The databases used in this book
- SQL `SELECT` and data manipulation syntax

INTRODUCTION

In this chapter, you learn the definition of programming and procedural languages, what PL/SQL is and why you need it, basic application models, locating documentation, SQL and PL/SQL tools, and the sample databases used throughout this book. A brief review of SQL statement syntax is included at the end of the chapter to make sure you're prepared to use SQL statements in the PL/SQL coding throughout this book. It's only a brief review to serve as a refresher; it doesn't cover all the facets of SQL you should already be familiar with, including queries, table creation, data manipulation, and transaction control. If you aren't well versed in these topics, be sure to review them before moving on to Chapter 2. After this chapter, you begin programming with PL/SQL.

APPLICATION PROGRAMMING AND PL/SQL

At its simplest, a **programming language** converts users' actions into instructions a computer can understand. All programming languages share some basic capabilities, such as manipulating data, using variables to hold data for processing, and making code reusable.

Structured Query Language (SQL) is a programming language that enables you to perform actions in a database, such as querying, adding, deleting, and changing data. However, it isn't a **procedural language** that programmers use to code a logical sequence of steps for making decisions and instructing a computer to perform tasks. For that purpose, you need a procedural language, such as Oracle PL/SQL. Oracle considers PL/SQL to be a procedural language extension of SQL, hence the "PL" in the name.

So what is PL/SQL? What role does it play in application programming? Why does a programmer need to learn it? These are probably some of the questions you have right now, and they need to be answered before you dive into Oracle programming. These answers should give you some insight into what you can accomplish with PL/SQL. The following section starts you off by covering general application programming concepts.

Application Programming

To clarify what's meant by "application programming," say you're an employee of a company named Brewbean's that retails coffee products on the Internet, and it needs to develop a software application to support the business. One part of the application consists of the user interface that customers use to access the product catalog and place an order. Figure 1-1 shows an example of this type of interface.

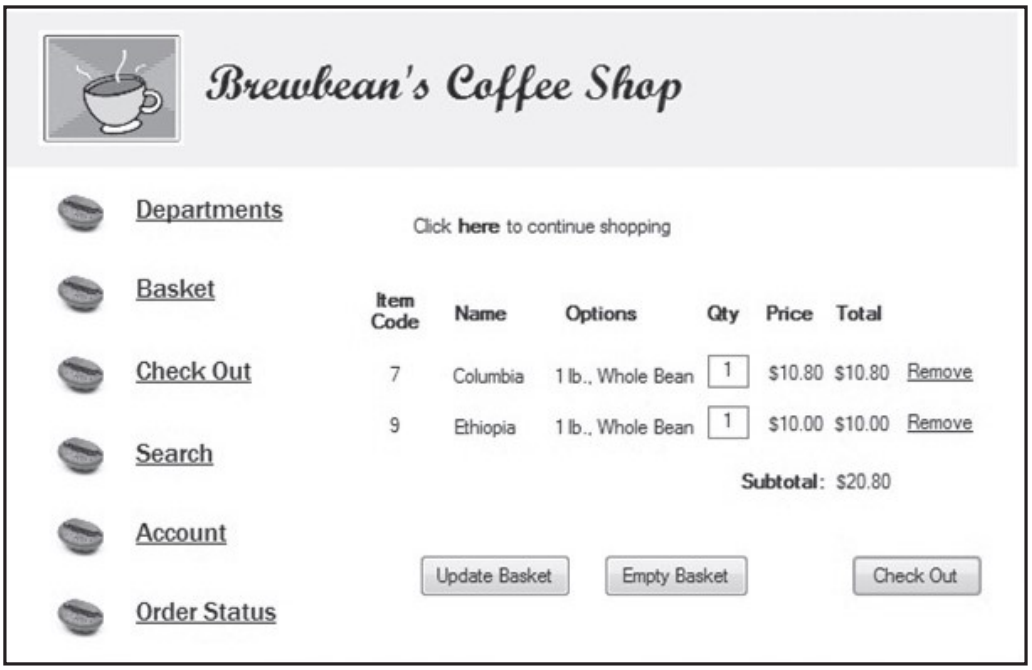


FIGURE 1-1 Brewbean's Coffee Shop shopping basket

This window displays the shopping basket containing the items a shopper has selected so far. The shopper has a number of choices at this point, such as continue shopping, change item quantities, remove items from the basket, or check out and complete the order.

Think of the potential processing this application might require. If, for example, the Check Out button is clicked, what should happen in the application? Possible processing activities include the following:

- Verify that the quantity ordered is greater than zero for each item.
- Calculate taxes.
- Calculate shipping charges.
- Check and/or update product inventory.
- Determine whether the shopper already has credit card information stored.

Behind the user interface in Figure 1-1 is where a programming language instructs the computer on what to do in response to a user's action, such as clicking the Check Out button. When the user clicks this button, several things might need to happen. First, the application needs to check the completeness of (or validate) the user's information. For example, you might need to verify that every line item has a quantity entered onscreen. You might also want to check the database to make sure all items are in stock. This is where PL/SQL enters the scene.

A procedural language makes it possible for developers to perform decision-making logic, such as `IF-THEN` conditions, in their applications. One example might be conditional statements that determine the shipping cost based on the total number of items ordered. If the total number of items is from three to five, the shipping cost equals $6; if the total number of items is more than five, the shipping cost equals $10; and so on.

PL/SQL Advantages

PL/SQL was modeled after Ada, a programming language built for the U.S. Department of Defense, and was considered conceptually advanced for the time. In the late 1980s, Oracle recognized a need for not only expanding functionality in its database system, but also for improving application portability and database security. At that point, PL/SQL was born and has been improved with every Oracle database release. PL/SQL is a proprietary language of Oracle, and a PL/SQL processing engine is built into the Oracle database and some developer tools. The following list summarizes a few advantages PL/SQL offers when working with an Oracle database:

- *Tight integration with SQL*—You can use your existing knowledge of SQL because PL/SQL supports SQL data manipulation, transaction control, functions, retrieving data with cursors, operators, and pseudocolumns. In addition, PL/SQL fully supports SQL data types, which reduces the need to convert data passed between your applications and the database.
- *Improved performance*—First, PL/SQL allows sending blocks of statements to Oracle in a single transmission. This is important in reducing network communication between your application and Oracle, especially if your application performs numerous database transactions. PL/SQL blocks can be used to group SQL statements before sending them to Oracle for execution. Otherwise, each SQL statement must be transmitted separately. Second, PL/SQL code modules (stored program units, described later in this section) are stored in executable form, which makes procedure calls efficient. Third, executable code is automatically cached in memory and shared among users. This can speed processing tremendously with a multiuser application that has repeated calls to modules of code. Fourth, a PL/SQL engine is embedded in

Oracle developer tools so that PL/SQL code can be processed on the client machine. This feature reduces network traffic.

- *Increased productivity*—PL/SQL can be used in many Oracle tools, and the coding is the same in all. Therefore, you can use your PL/SQL knowledge with many development tools, and the code you create can be shared across applications. Because a PL/SQL engine is part of the server, these code modules can also be used or called from almost any application development language. You can use Visual Basic or Java to develop an application but still harness the power of PL/SQL with Oracle.
- *Portability*—PL/SQL can run on any platform that Oracle can run. This feature is important so that developers can deploy applications on different platforms easily.
- *Tighter security*—Database security can be increased with application processing supported by PL/SQL stored program units, which give users access to database objects without having to be granted specific privileges. Therefore, users can access these objects only via PL/SQL stored program units.

NOTE

A database administrator (DBA) can automate and handle some tasks more easily by using the power of PL/SQL. Many PL/SQL scripts developed to handle DBA-type tasks are available free in books, Web sites, and user groups, so being familiar with the PL/SQL language is quite beneficial for DBAs.

One key advantage is that program units can be stored in the Oracle database and called from your development tool. Processing SQL statements stored in the database can be more efficient than processing those stored in application code. Statements stored in application code must be transmitted to the database server to be processed. PL/SQL program modules stored in the database are referred to as **stored program units**.

NOTE

If you have a piece of code that might be used by a variety of applications, saving it on the server allows several applications to share it. In addition, Oracle has built a PL/SQL engine into a number of its developer tools, such as Oracle Forms, so Oracle developers can code an entire application—from client screen logic to database manipulation—with PL/SQL.

APPLICATION MODELS

An application model is a general framework or design that describes the structure of an application's components in terms of where the supporting software is located. An application model has three main components:

- *User interface*—The windows displayed to users to enter information or take actions, such as clicking a button. The user interface component can be developed with tools such as Visual Basic, Java, or Oracle Forms.

- *Program logic*—The programming code that provides the logic of what the application does. PL/SQL handles this component, which can be considered the "brains" of an application.
- *Database*—The database management system providing the physical storage structure for data and mechanisms to retrieve, add, change, and remove data. The Oracle server provides this component.

To see more clearly where PL/SQL is used in an application, take a closer look at basic two-tier and three-tier application models, in which each tier manages different application components.

NOTE

The traditional two-tier application model is also called a "client/server application." Both terms are used throughout this book.

Two-Tier or Client/Server Application Model

In a client/server model, shown in Figure 1-2, an executable program or application is loaded on the user's computer. The application contains the user interface and some programming logic. Some processing can take place on the client side (the user's computer). For instance, the processing might be verifying that information has been entered in required fields. Other processing might require transmitting requests, such as an SQL statement to query requested data, to the database server.

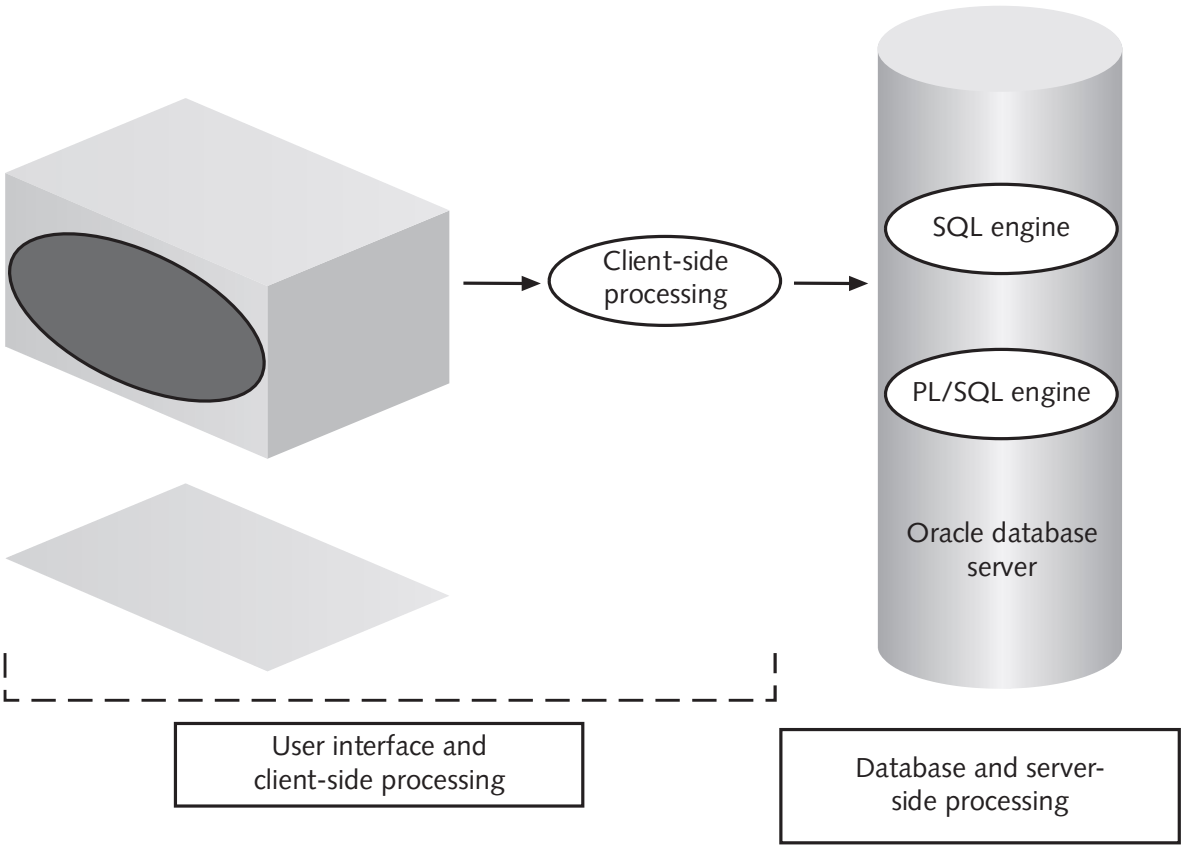


FIGURE 1-2 The client/server application model

If you're using non-Oracle developer tools, such as Visual Basic, client-side code can be VBScript that includes calls to PL/SQL program units stored on the Oracle server. However, if you're using Oracle development tools, such as Oracle Forms, a PL/SQL engine exists on the client as well as on the database server, so all application coding is done with PL/SQL.

With Oracle development tools, PL/SQL code resides on both the client side and server side. That is, PL/SQL is saved as part of the Oracle Forms application logic on the client side and stored as named program units on the database server. A **named program unit** is simply a block of PL/SQL code that has been named so that it can be saved (stored) and reused. The term "stored" indicates that the program unit is saved in the database and, therefore, can be used or shared by different applications. Table 1-1 describes each type of program unit briefly. You create all these program units in this book.

TABLE 1-1 Stored Program Unit Types

Stored Program Unit Type	Description
Procedure	Performs a task. Can receive and return multiple values.
Function	Performs a task and typically returns only one value. Within certain parameters, it can be used in SQL statements.
Database trigger	Performs a task automatically when a data manipulation language (DML) action occurs on the associated table or system event.
Package	Groups related procedures and functions, which makes additional programming features available.

Procedures, functions, triggers, and packages integrated in an Oracle Forms application are considered client-side program units. They're referred to as application program units rather than stored program units. An example of an application trigger is PL/SQL code that runs automatically when a button is clicked onscreen. These events are handled in Oracle Forms development, which is beyond the scope of this book.

Three-Tier Application Model

The three-tier model has become more widely used because it attempts to reduce application maintenance and allows supporting more users. In this model, shown in Figure 1-3, the user interface is typically displayed in a Web browser and often referred to as a "thin client." Unlike the two-tier model, application code isn't loaded on the client machine; it's stored on an application server, also referred to as the "middle tier." This model has been critical in supporting the explosion of mobile applications.

This model's three tiers are the user interface, the application server, and the database server. The Oracle application server allows deploying Oracle Forms applications via the Web and contains the user interface and processing logic, which together respond to user actions and send code to the database server for processing.

NOTE

In either application model, PL/SQL's role is the same: to provide the logic for instructing the computer what to do when an event occurs. (An **event** can range from a user action, such as clicking a button, to a table update statement that calls a database trigger automatically.) In this book, all code is placed server side as stored program units.

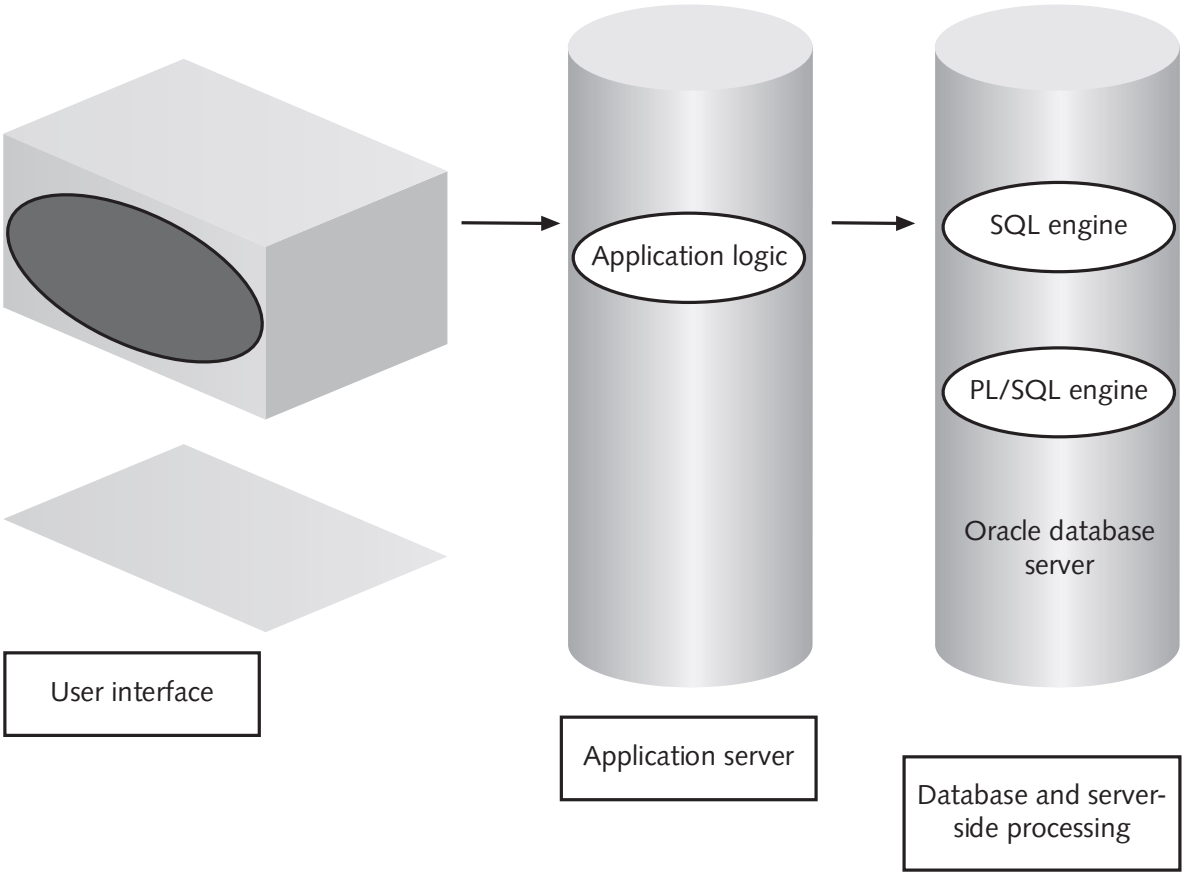


FIGURE 1-3 A three-tier application model

ORACLE RESOURCES

As you have probably discovered, numerous books are available on Oracle system topics—including PL/SQL. This section gives you information on Web resources for learning the PL/SQL language to complement the information in this book.

Web Resources

Oracle's Web site is called the Oracle Technology Network (OTN) and offers a variety of useful resources, including documentation, white papers, downloads, and discussion forums. You can access it from Oracle's home page (*www.oracle.com*) or by going directly to *http://otn.oracle.com*. You must log in to access certain areas of this Web site, but membership is free.

The OTN site is updated often, but the home page always lists categories of information to help you find what you're looking for. For example, the home page includes Documentation, Discussion Forums, Sample Code, and Learning Library links. You can click the Documentation link and follow the links to arrive at the Database Library page for the Oracle version you're interested in (for this book, Oracle Database 11g Release 2 [11.2]). Then use the Master Book List link to view all online books, including PL/SQL Language Reference (useful for the topics in this book) and PL/SQL Packages and Types Reference. A SQL Language Reference is also available.

The OTN Web site also has pages on specific technology topics, such as Oracle Database 11g XE, Database Focus Areas, and Application Development. After selecting a database version, click Database Focus Areas and then Application Development to arrive at a page with many useful developer resources. In the Languages section, click the link for PL/SQL to

find the page shown in Figure 1-4. In the Tools section, click the SQL Developer link to find more detailed information on the development tool used in this book.

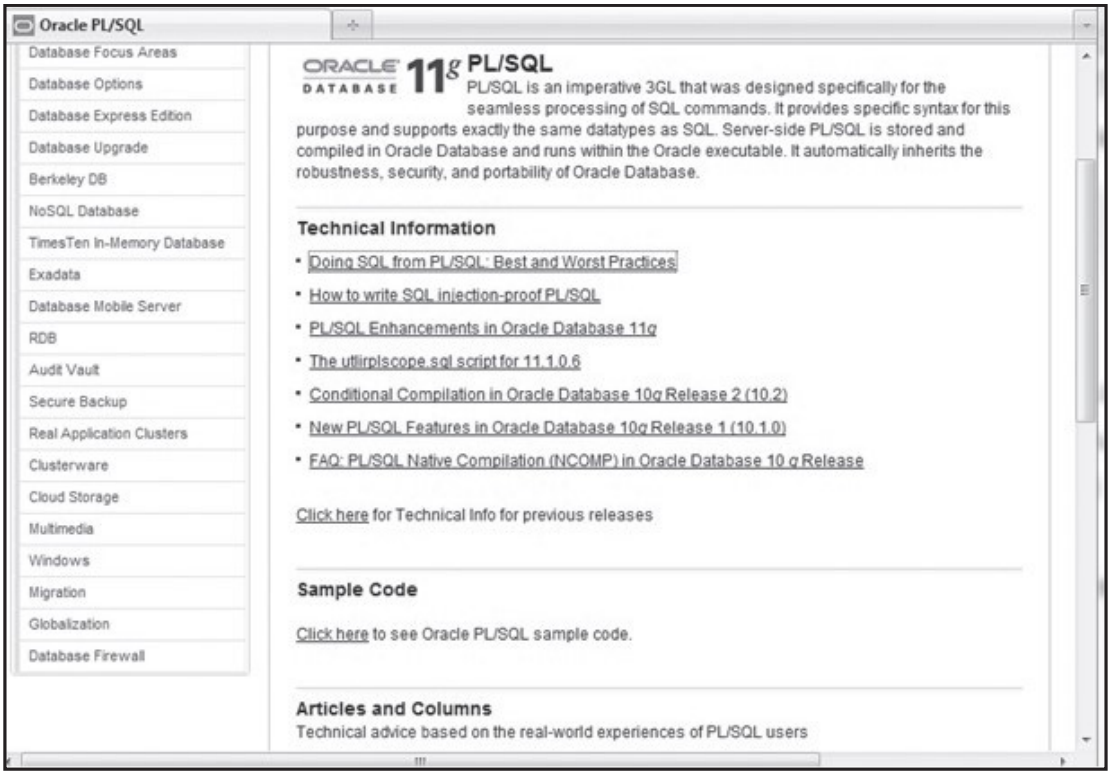


FIGURE 1-4 The PL/SQL page on the OTN site

Doing an Internet search on PL/SQL leads to a variety of Web pages with helpful information, such as PL/SQL Obsession (*www.toadworld.com/SF/*) by Steven Feuerstein, a leading expert in PL/SQL. Although these pages are useful, being familiar with the language basics first is usually better.

SQL AND PL/SQL TOOLS

Wading through the many software tools that Oracle offers can get confusing. In addition, you'll soon learn that many third-party software companies offer tools for Oracle developers. This section introduces the Oracle tools used in this book for PL/SQL development and lists some third-party tools that are popular in the Oracle development community. Note that this list isn't exhaustive, nor does it give recommendations. It merely attempts to acquaint you with some options for selecting tools.

NOTE

Your instructor is a valuable resource in helping you decide which tools are best for you.

Software Tools for This Book

This book covers the newest version of the Oracle database: Oracle 11*g*. However, this book can also be used with Oracle 10*g*. PL/SQL version numbering has matched the database version numbering starting with Oracle 8, so PL/SQL 11*g* is used in this book to

complement the Oracle 11g database. Note, however, that the major enhancements in PL/SQL versions were mainly internal database engine changes made to improve the performance of PL/SQL processing. Any features that are new in Oracle 11g are noted as such. The following sections describe the two free PL/SQL developer tools from Oracle, SQL*Plus and SQL Developer.

SQL*Plus

Starting with Oracle 11g, only the console or command-line SQL*Plus interface is shipped with the database product. SQL*Plus is a basic tool included with the Oracle server that allows users to send SQL and PL/SQL statements directly to the Oracle database server for processing. Figure 1-5 shows the command-line SQL*Plus interface with a simple query entered. Many users don't find this interface as easy to use as SQL Developer, but Oracle users should be familiar with it because it's available with all Oracle installations.

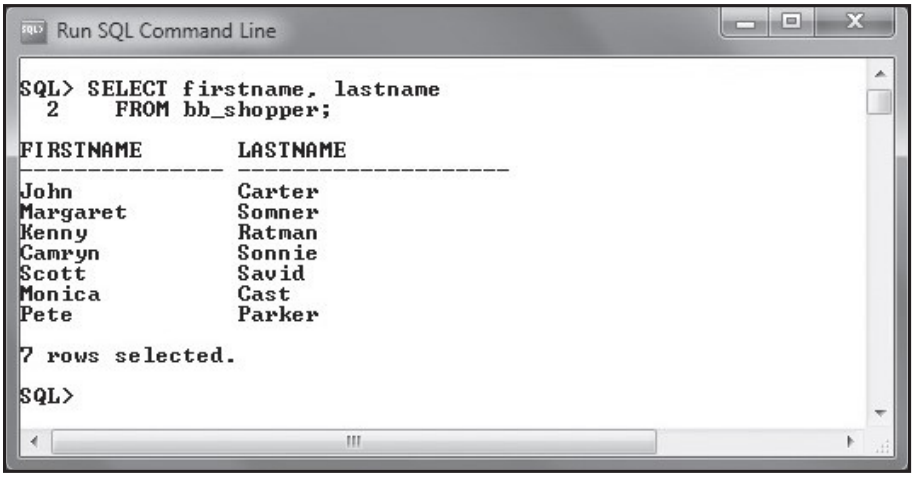


FIGURE 1-5 The SQL*Plus interface

NOTE

This book uses SQL Developer for examples and screenshots. However, you can run all the sample code in SQL*Plus. See Appendix B for more details on SQL*Plus and Oracle database installation.

Oracle SQL Developer

Oracle SQL Developer is available free on the Oracle Web site. This utility has a graphical user interface (GUI), which makes it easier for developers to explore database objects, review and edit code quickly, and create and debug program units. Figure 1-6 shows the main window of Oracle SQL Developer.

Even though you can run PL/SQL code in SQL*Plus, Oracle SQL Developer offers features that simplify development tasks, such as color-coding syntax and using breakpoints to assist in debugging code. (Breakpoints are stopping points during execution that enable programmers to evaluate variable values while code is running.)

In practice, SQL*Plus, Oracle SQL Developer, and third-party tools are used to test and debug stand-alone PL/SQL program units. Examples in this book are shown in the SQL Developer interface.

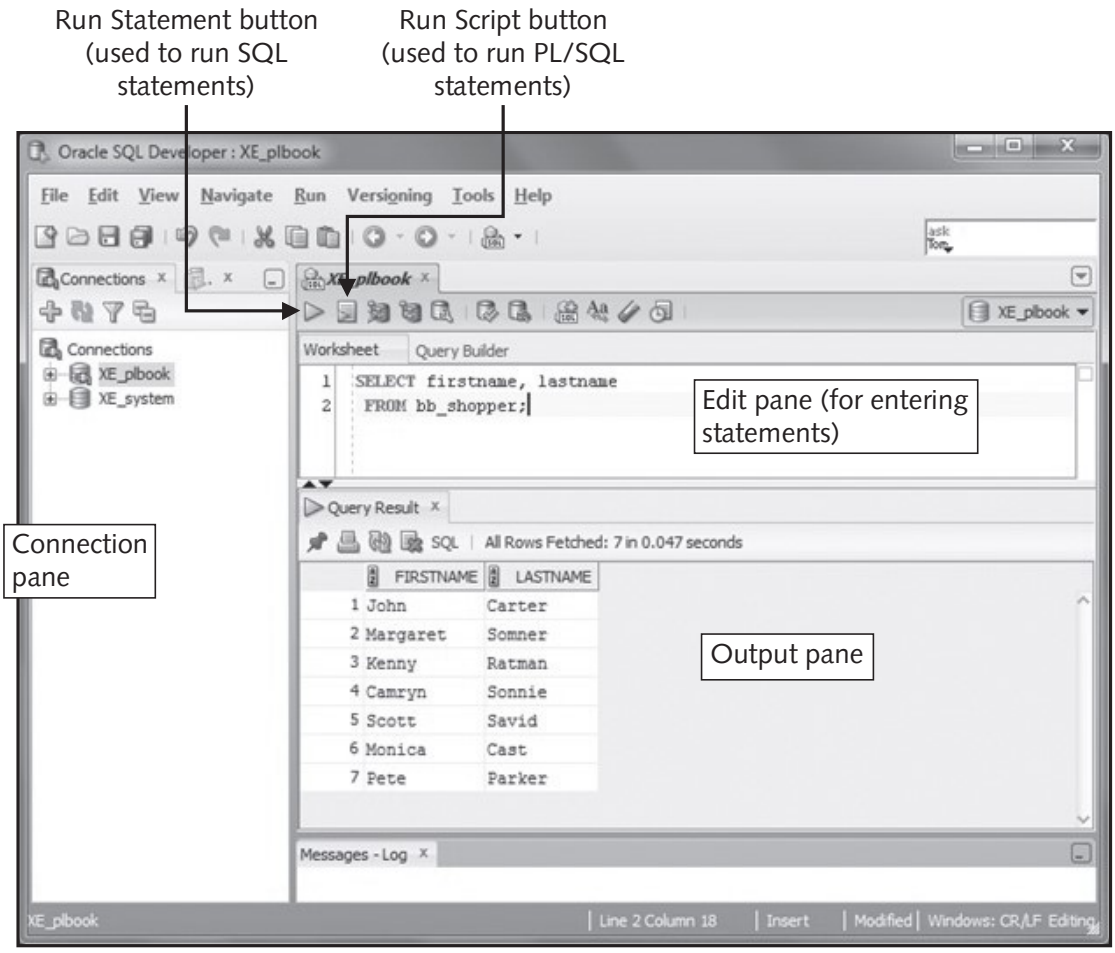


FIGURE 1-6 The Oracle SQL Developer main window

NOTE

Appendix B has instructions for Oracle SQL Developer installation.

Working with the Tools

To do the coding activities in this book, you need access to the Oracle 11*g* database and SQL Developer. In addition, the appendixes use SQL*Plus and Tool for Oracle Application Developers (TOAD), which is a third-party tool. If you're working in a computer lab setting, the correct software should already be installed for you.

If you want to install Oracle 11*g* on your own computer, follow the installation instructions for Oracle Database 11*g* Express Edition in Appendix B. For learning purposes, many programmers install Oracle Express, which is more suitable for desktop installations and is available free for learning and testing. It's offered on the disc accompanying this book, too.

Third-Party PL/SQL Development Tools

Many other companies have developed software for PL/SQL developers. As you progress through this book and become familiar with creating PL/SQL program units, you might want to take some time to experiment and compare some tools from third parties. Table 1-2 lists

some popular tools and their Web sites for your reference. Many companies offer a free trial download for testing products.

TABLE 1-2 Third-Party PL/SQL Development Tools

Tool Name	Web Site
TOAD	*www.quest.com*
SQL Navigator	*www.quest.com*
PL/SQL Developer	*www.allroundautomations.com*

NOTE

TOAD by Quest Software is one of the most popular third-party Oracle tools and is explained briefly in Appendix C.

DATABASES USED IN THIS BOOK

Programmers soon realize that to become successful application developers, they must understand databases thoroughly. Therefore, be sure to review this section carefully to gain a good understanding of the design of databases used in this book. The SQL review in the next section also helps you become familiar with this book's databases.

The main database used in this book supports a company retailing coffee via the Internet, over the phone, and in one walk-in store. The company, Brewbean's, has two main product categories: coffee consumables and brewing equipment. It also hopes to add a coffee club feature to entice return shoppers. The Brewbean's database is referenced throughout chapter material and used in Hands-On Assignments at the end of each chapter. Two additional databases are used in end-of chapter exercises. A donor database that tracks donation pledges and payments is used in Hands-On Assignments, and a database involving a movie rental company is used in Case Projects. The next sections introduce these three databases.

The Brewbean's Database

Figure 1-7 shows the basic entity-relationship diagram (ERD) for the Brewbean's database pertaining to customer orders. An ERD serves as a visual representation of a database.

Note that all table names for the Brewbean's database start with the prefix BB_. The BB_DEPARTMENT table lists the two main business areas in the company (coffee sales and equipment sales), and the BB_PRODUCT table contains information on all products, including name, description, price, and sales pricing.

Three tables are used to manage product options. The BB_PRODUCTOPTION CATEGORY table lists main categories for products, such as size and form. The BB_PRODUCTOPTIONDETAIL table identifies choices in each category (for example, whole bean or ground for form). The BB_PRODUCTOPTION table links each product to its applicable options. Each product can be associated with many options, so there's a one-to-many relationship between the BB_PRODUCT and the BB_PRODUCTOPTION tables.

The BB_SHOPPER table serves as the focal point for identifying customers and contains customers' names, addresses, e-mail addresses, and logon information. When a

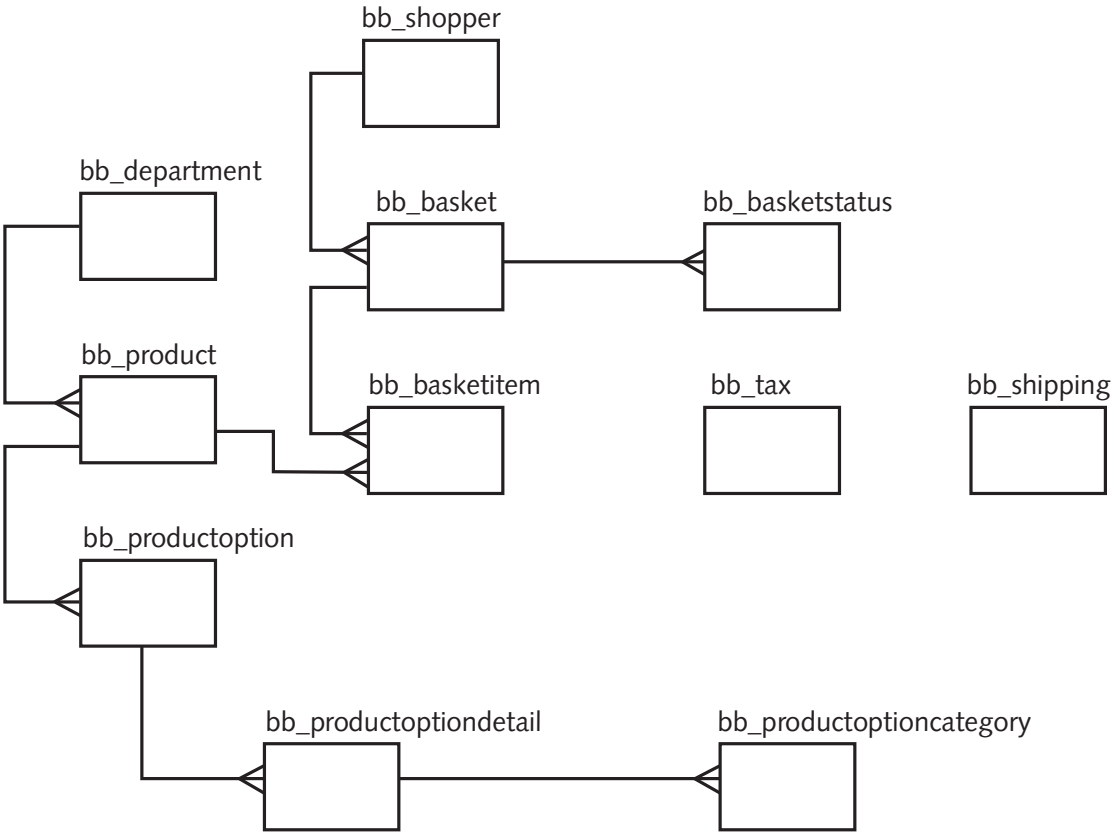


FIGURE 1-7 The Brewbean's database ERD

customer begins shopping, a new basket is created in the BB_BASKET table. This table is one of the largest in the database; it holds order summary, shipping, and billing information. As the shopper selects items, they're inserted in the BB_BASKETITEM table, which holds shopper selections by basket number.

The BB_BASKETSTATUS table stores data related to order status. Each status update is recorded as a new row in this table. Possible statuses include order placed, order verified and sent to shipping, order shipped, order canceled, and order on back-order. The other tables associated with completing an order include BB_TAX and BB_SHIPPING. The company currently calculates shipping based on the quantity of items ordered.

Appendix A lists data by table for your reference. Take some time now to review this information and become familiar with the database.

Creating the Database

To create the Brewbean's database, you need to run a script file containing a series of SQL statements that build all the necessary objects and insert data. Follow these steps to run the script:

1. Make sure you have the `c1_BBcreate.sql` file in the Chapter01 folder. This file contains the script for creating the database.
2. Start SQL Developer and connect to the Oracle database.
3. Click **File**, **Open** from the menu.

4. Click the **c1_BBcreate.sql** file, and then click **Open**. The code in this script is displayed in the Edit pane of SQL Developer. Scroll through the SQL statements; they should look familiar.
5. Run the script by clicking the **Run Script** button on the Edit pane toolbar. If necessary, click **OK** in the Select Connection dialog box. Figure 1-8 shows the results.

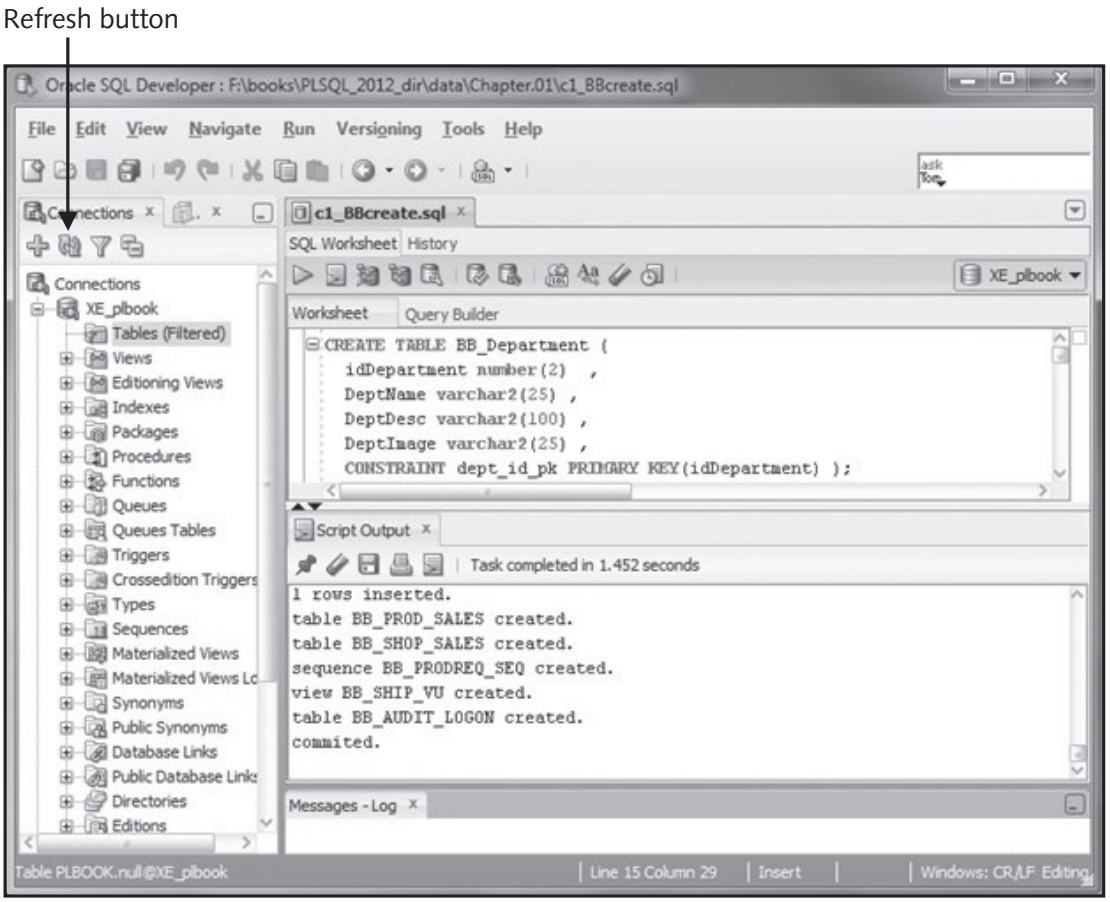


FIGURE 1-8 Creating the Brewbean's database

6. Scroll through the Script Output pane at the bottom to review results of the statements. If you have any errors caused by insufficient privileges, ask your instructor for assistance.
7. In the Connections pane on the left, expand the **Tables** node to view the tables that were created. You might need to click the **Refresh** button at the top of the Connections pane to see these objects.

The DoGood Donor Database

The DoGood Donor database, used in Hands-On Assignments at the end of each chapter, is used to track donors, donation pledges, and pledge payments. Figure 1-9 shows a basic ERD for this database.

The DD_DONOR table is the core of this database because all other data depends on the donor, who represents a person or company that has committed to make a donation to the DoGood organization. Donor identification data is stored in the DD_DONOR table, and

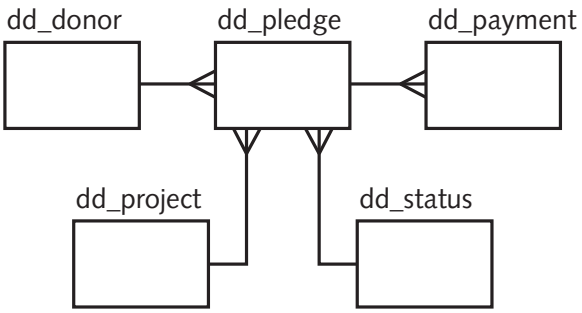


FIGURE 1-9 The DoGood Donor database ERD

donation commitments or pledges are recorded in the DD_PLEDGE table. Two important options are indicated by data in the DD_PLEDGE table: PAY_MONTHS indicates the number of equivalent monthly payments needed to complete the pledge commitment (a 0 indicates a lump sum payment), and PROJ_ID indicates the project for the pledge amount. A pledge can be unassigned or not dedicated to a specific project.

The DD_PROJECT table is the master list of all projects to which pledges can be dedicated. The DD_STATUS table defines the pledge status values assigned to each pledge in the DD_PLEDGE table, indicating whether the pledge is still active, complete, overdue, or closed without full collection. The DD_PAYMENT table stores data for each pledge payment.

Create the DoGood Donation database by using the same steps as for the Brewbean's database in the previous section. Click the `c1_DDcreate.sql` file in the Chapter01 folder and run this script.

The More Movies Database

To work with the second Case Project at the end of each chapter, you need to build a database that supports a movie rental company named More Movies. In these projects, you create application components to support membership and renting processes for More Movies. Figure 1-10 shows this database's ERD.

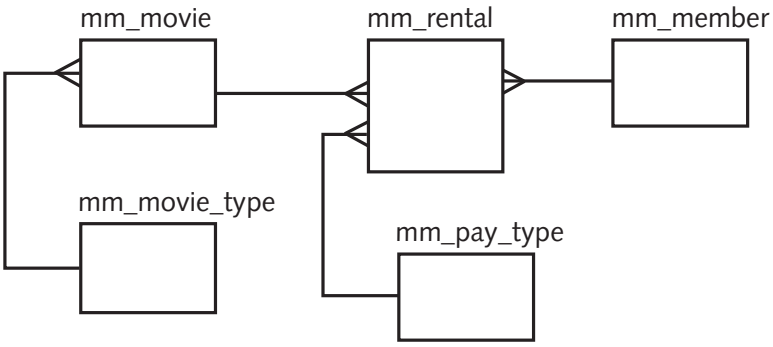


FIGURE 1-10 The More Movies database ERD

The MM_MEMBER table is the master customer list, and the MM_MOVIE table holds information for the movie inventory. All movie rental transactions are tracked with the MM_RENTAL table. The MM_MOVIE_TYPE table is a list of movie genres, and the MM_PAY_TYPE is a list of payment methods used for rentals.

To create the More Movies database, follow the same steps used to create the Brewbean's database, but click the `c1_MMcreate.sql` file in the Chapter01 folder and run this script.

SQL QUERY REVIEW

Before you begin learning PL/SQL coding, it's important to be familiar with SQL syntax. As mentioned, SQL statements are embedded in PL/SQL modules to perform database interaction tasks. This section includes examples of SQL queries, table creation, data manipulation, and transaction control statements.

The following brief review helps you recall the syntax of SQL statements. If you find you don't understand certain concepts, go back to what you learned about SQL statements in previous courses. You need to be familiar with queries (`SELECT`), DML statements (`INSERT`, `UPDATE`, `DELETE`), data definition language (DDL) statements (`CREATE`, `ALTER`, `DROP`), and transaction control statements (`COMMIT`, `ROLLBACK`).

First, review the basic syntax of a `SELECT` statement:

```
SELECT <columns>
FROM <tables, views>
WHERE <conditions>
GROUP BY <columns>
HAVING <aggregation conditions>
ORDER BY <columns>;
```

Try working through a few examples to make sure you're familiar with the coding techniques. Using the Brewbean's database built earlier in this chapter, what statement could be used to produce a list of all products displaying the product name, active status, and department name? Notice that both the BB_PRODUCT and BB_DEPARTMENT tables are needed to retrieve the required information. You need to join these two tables by using a traditional join in the `WHERE` clause (see Figure 1-11) or an ANSI join in the `FROM` clause (see Figure 1-12). Note that the `ON` keyword could be used instead of the `USING` clause. In addition, the `INNER` keyword could be left out.

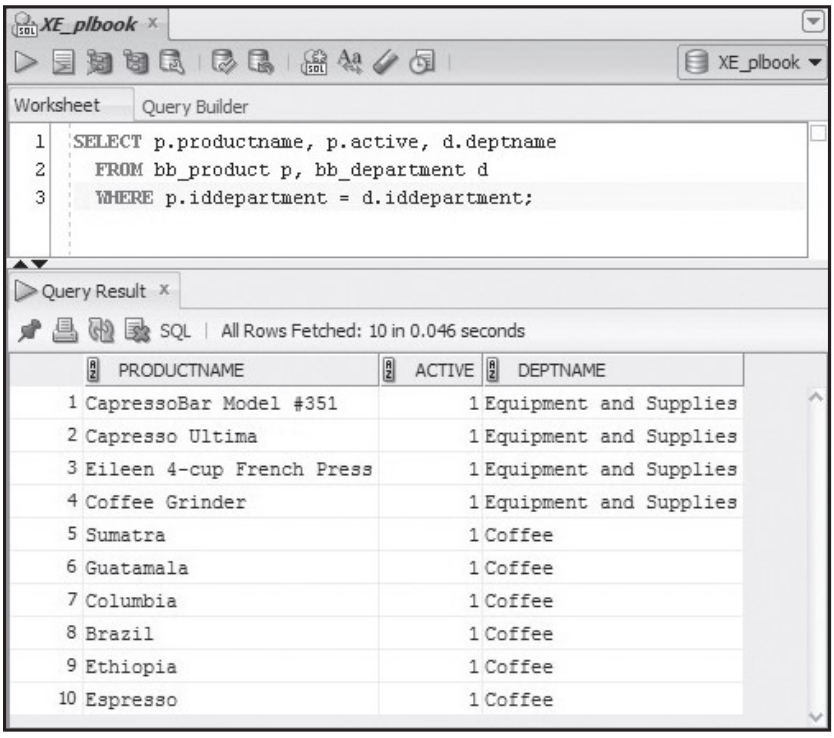


FIGURE 1-11 A query with a traditional join

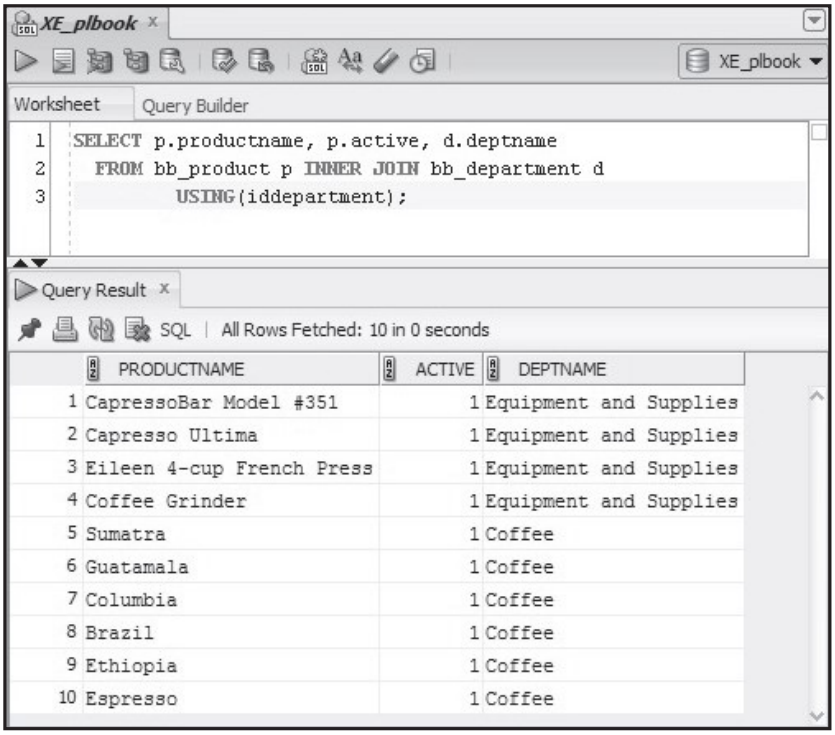


FIGURE 1-12 A query with an ANSI join

Notice the use of table aliases assigned in the `FROM` clause. An alias serves as a nickname to reference a table when qualifying columns.

Next, try a query with an aggregate function. What statement is needed to produce a list showing the number of products in the database by department name? Figure 1-13 displays the statement using the aggregate `COUNT` function.

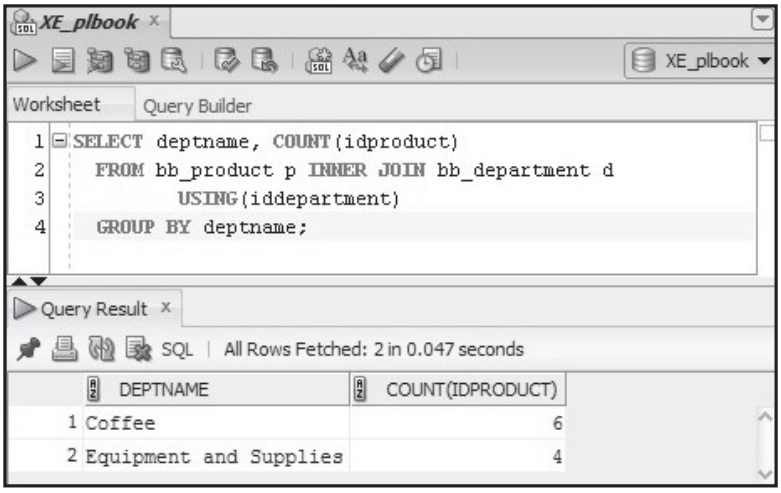


FIGURE 1-13 A query with an aggregate function

TIP

What if you want only departments with five or more products listed? A `HAVING` clause must be added to check whether the count is five or more. Try it!

Now try data-filtering techniques. Write a statement that produces the average price of coffee products. Review the BB_PRODUCT table data, and be sure to look at the data in the TYPE column. Figure 1-14 shows the statement using the aggregate AVG function and a WHERE clause to include only coffee products.

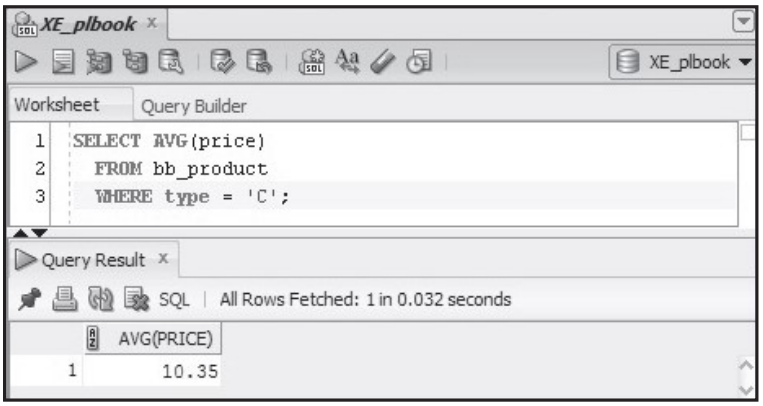


FIGURE 1-14 A query with a WHERE clause

TIP

What if the TYPE value could be stored in uppercase or lowercase characters? You can use the UPPER or LOWER function in the WHERE clause to eliminate the question of case sensitivity.

The preceding examples focus on querying data from an existing database. Next, you review commands used to create tables and manipulate table data. You create a table that holds information for automobiles available at a car rental agency. Build and run the statement shown in Figure 1-15 to create the table. Notice that a primary key constraint is included. Assigning constraint names is helpful because they're referenced if a constraint error occurs in a data manipulation operation.

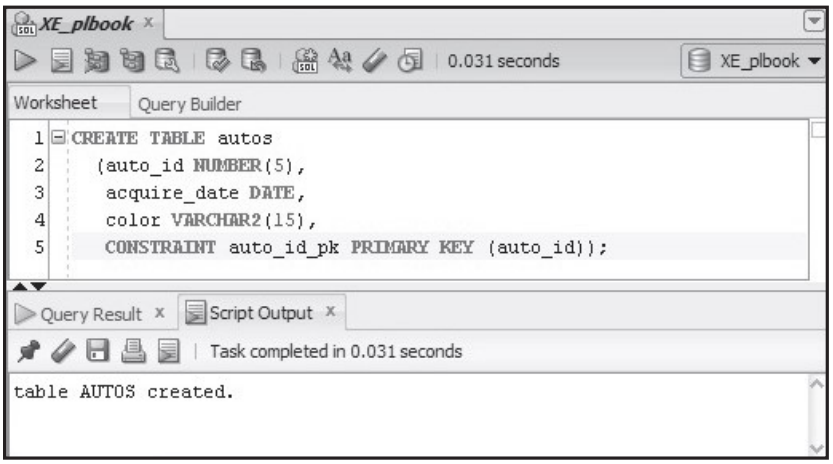


FIGURE 1-15 Creating a table

How can you verify what tables exist in your schema? In SQL Developer, you can simply expand the Tables node in the Connections pane and select the AUTOS table to view its structure. (Keep in mind that you might need to refresh the Connections pane.) However, you can also query the data dictionary view of USER_TABLES to list the tables

in your schema. How can you check a table's structure with a command? Figure 1-16 shows the command for this task.

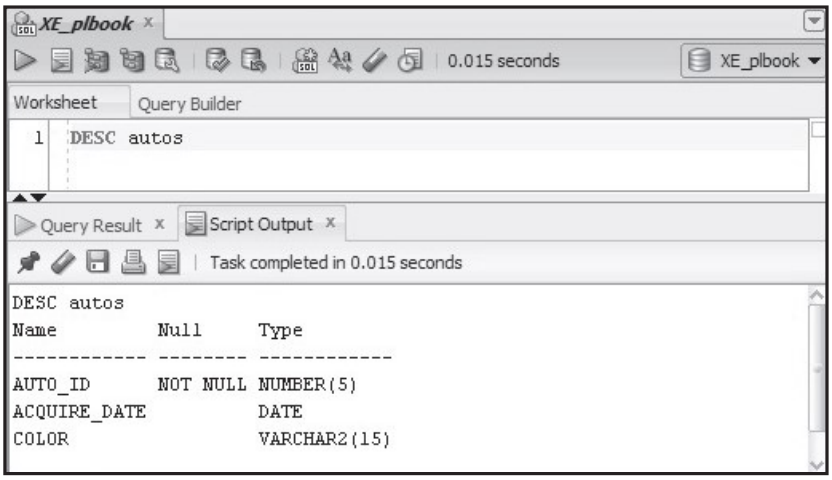


FIGURE 1-16 Listing the structure of a table

Now that you have a table, you can use DML statements to add, modify, and delete data in the table. You can also use transaction control statements to save or undo DML actions. First, issue the two `INSERT` statements shown in Figure 1-17 to add two rows to the table. The `COMMIT` statement that follows saves the data rows permanently. Finally, issue the `SELECT` query on the last line to verify that the data has been inserted.

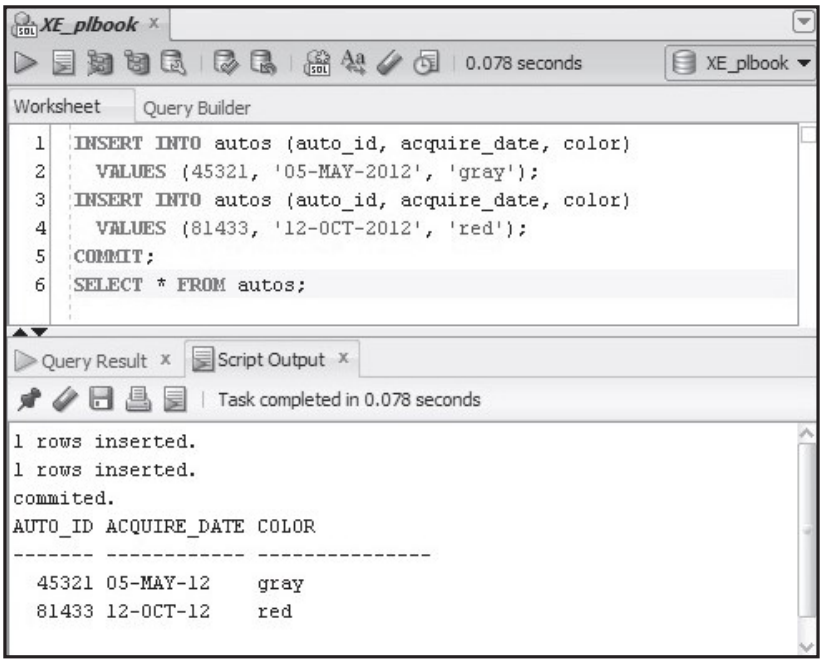


FIGURE 1-17 Adding rows of data

TIP

Four separate SQL statements are used in Figure 1-17. Select each statement, and click the Run Statement button on the Worksheet pane's toolbar to run each statement separately and view the results. If you use the Run Script button, all statements run at once.

NOTE

A sequence is typically used to provide primary key column values. A call of *sequence_name* NEXTVAL is used in an INSERT statement to retrieve a value from a sequence.

The UPDATE statement can be used to modify existing data in the database. Write and run a statement to change the color of the auto with the ID 45321 to silver. Be sure to include a WHERE clause, as shown in Figure 1-18, to ensure that only one row is updated. Keep in mind that omitting a WHERE clause means all rows are updated. Finally, issue a query to verify the modification.

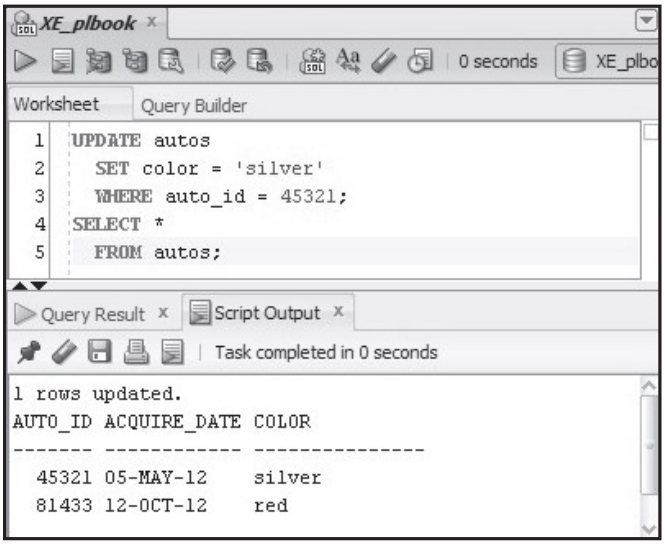


FIGURE 1-18 Modifying a row

Say the previous change was in error, and you want to undo the UPDATE. Because this action hasn't been saved permanently with a COMMIT statement, a ROLLBACK action can be issued to undo the UPDATE statement. Issue a ROLLBACK statement, and query the data to verify that the change has been reversed (see Figure 1-19).

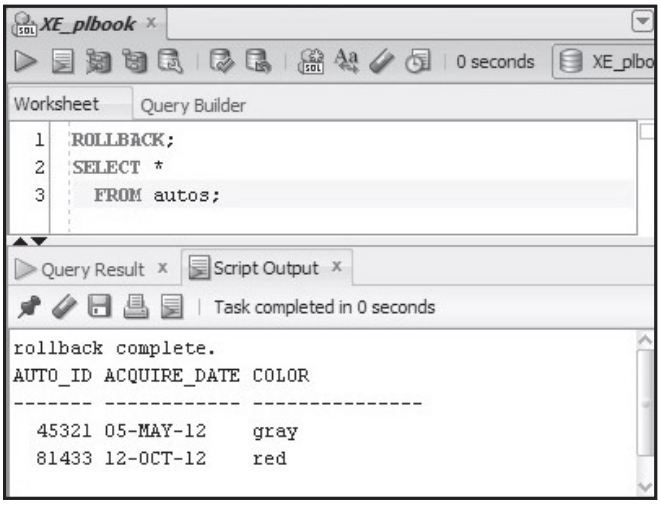


FIGURE 1-19 Using ROLLBACK to undo a DML action

To remove a row, you use the DELETE command. Issue a statement to remove the row of data for the auto with the ID 45321 (see Figure 1-20). Again, notice that the WHERE clause determines which rows are affected by the statement. To save this change permanently, you would need to issue a COMMIT statement.

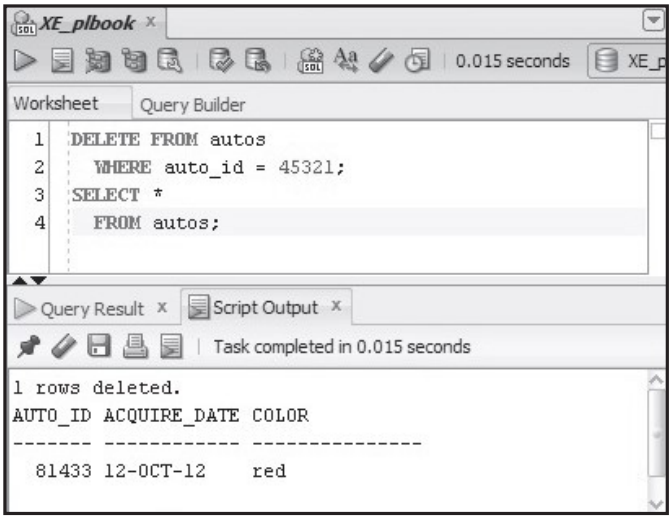


FIGURE 1-20 Using DELETE to remove a row

Remove the table from the database by using the DROP TABLE statement shown in Figure 1-21. Notice that a query attempt on the table after it's been removed displays an error stating that the object doesn't exist.

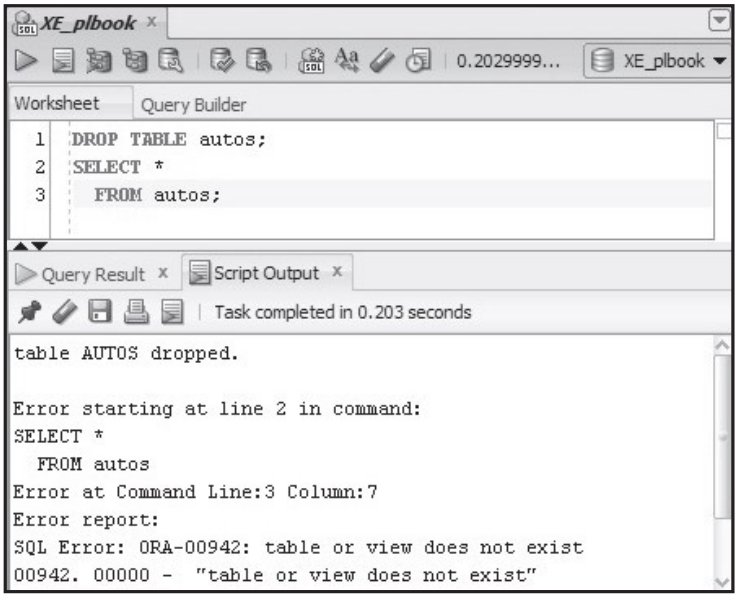


FIGURE 1-21 Removing a table

These examples should help you review the SQL statement syntax you have learned previously. Before you venture into PL/SQL coding, make sure you're thoroughly familiar with the following SQL query topics:

- Joins (traditional and ANSI)
- Restricting data with `WHERE` and `HAVING` clauses
- Single-row functions
- Group/aggregate functions
- Subqueries
- Views
- Sorting
- Tables (create, alter, drop, constraints)
- Sequences
- DML actions
- Transaction control

Chapter Summary

- Programmers can use a procedural language to code a logical sequence of steps for making decisions and instructing the computer on what tasks to perform.
- SQL is used for creating and manipulating a database, but it's not a procedural language.
- PL/SQL is a procedural language, and the Oracle database and software tools contain a PL/SQL processing engine.
- PL/SQL improves application portability because it can be processed on any platform Oracle runs on.
- Using PL/SQL can improve security because users don't need to be granted direct access to database objects.
- Applications contain three main components: a user interface, program logic, and a database.
- The two-tier or client/server application model splits programming logic between the client machine and the database server.
- The three-tier application model places much of the program code on an application server (the middle tier).
- Programming code on the client machine is referred to as client-side code, and code on the database server is referred to as server-side code.
- A named program unit is a block of PL/SQL code that has been saved and assigned a name so that it can be reused.
- A stored program unit is a named program unit that's saved in the database and can be shared by applications. Procedures and functions are named program units that are called to perform a specific task.
- A database trigger is PL/SQL code that's processed automatically when a particular DML action occurs. A package is a structure that allows grouping functions and procedures into one container.
- An application event is some activity that occurs, such as a user clicking an item onscreen, which causes some processing to occur.
- Documentation is available on the Oracle Technology Network (OTN) Web site.
- Both SQL*Plus and SQL Developer are Oracle software tools that allow submitting SQL and PL/SQL statements directly to the Oracle server. Oracle SQL Developer is a software tool for creating and testing program units.
- Many third-party software tools are available for developing PL/SQL program units.
- You need to review a database's ERD before you can work with the data effectively.
- You need to be familiar with SQL statement syntax before learning PL/SQL.

Review Questions

1. What application model typically displays the user interface in a Web browser?
   a. client/server
   b. two-tier
   c. three-tier
   d. thin-tier
2. Which of the following is *not* a type of stored program unit in PL/SQL?
   a. procedure
   b. application trigger
   c. package
   d. database trigger
3. The term "named program unit" indicates which of the following?
   a. The PL/SQL block is assigned a name so that it can be saved and reused.
   b. The PL/SQL block is of a certain type.
   c. The PL/SQL block is executable as an anonymous block.
   d. The PL/SQL block is saved client side.
4. Any application model usually represents which of the following basic components? (Choose all that apply.)
   a. user interface
   b. program logic
   c. coding style
   d. database
5. When working with an Oracle database, which of the following is considered an advantage of PL/SQL? (Choose all that apply.)
   a. tight integration with SQL
   b. easier naming conventions
   c. tighter security
   d. improved performance
6. Which of the following is a free GUI tool from Oracle for editing PL/SQL?
   a. SQL*Plus
   b. Oracle Developer
   c. Oracle SQL Developer
   d. Only third-party tools are available.

7. Which of the following is an Oracle tool included with the Oracle database server that allows sending SQL and PL/SQL statements to the server?
   a. SQL*Plus
   b. PL/SQL Builder
   c. PL/SQL Creator
   d. Procedure Builder
8. A procedural programming language allows including which of the following?
   a. decision-making logic
   b. inserts
   c. DML statements
   d. table creation statements
9. "Application portability" refers to the capability to _____ .
   a. upload and download
   b. create a small executable
   c. move to other computer platforms
   d. transmit data efficiently
10. A two-tier application model is commonly referred to as a(n) _____ application model.
    a. *n*-tier
    b. client/server
    c. double-layered
    d. user-database
11. Name the four types of stored program unit structures and the basic differences between them.
12. If you aren't using Oracle development tools, such as Oracle Forms, should you pursue learning PL/SQL? Why or why not?
13. Describe the major difference between a two-tier and a three-tier application model.
14. Describe what a user interface is and the role a procedural language plays in user interfaces.

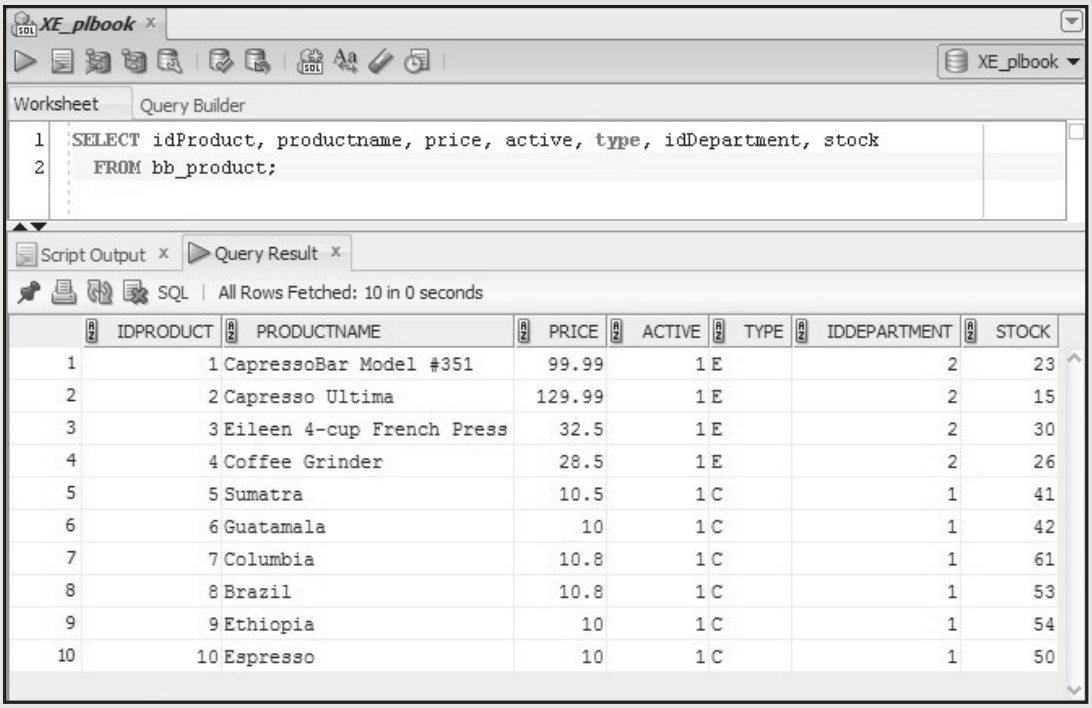
Hands-On Assignments Part I

Assignment 1-1: Reviewing SQL and Data in the Brewbean's Database

Becoming familiar with the Brewbean's database is important because it's used in examples and exercises throughout this book. For this assignment, you create a few queries to review the data.

First, you query the database by following these steps:

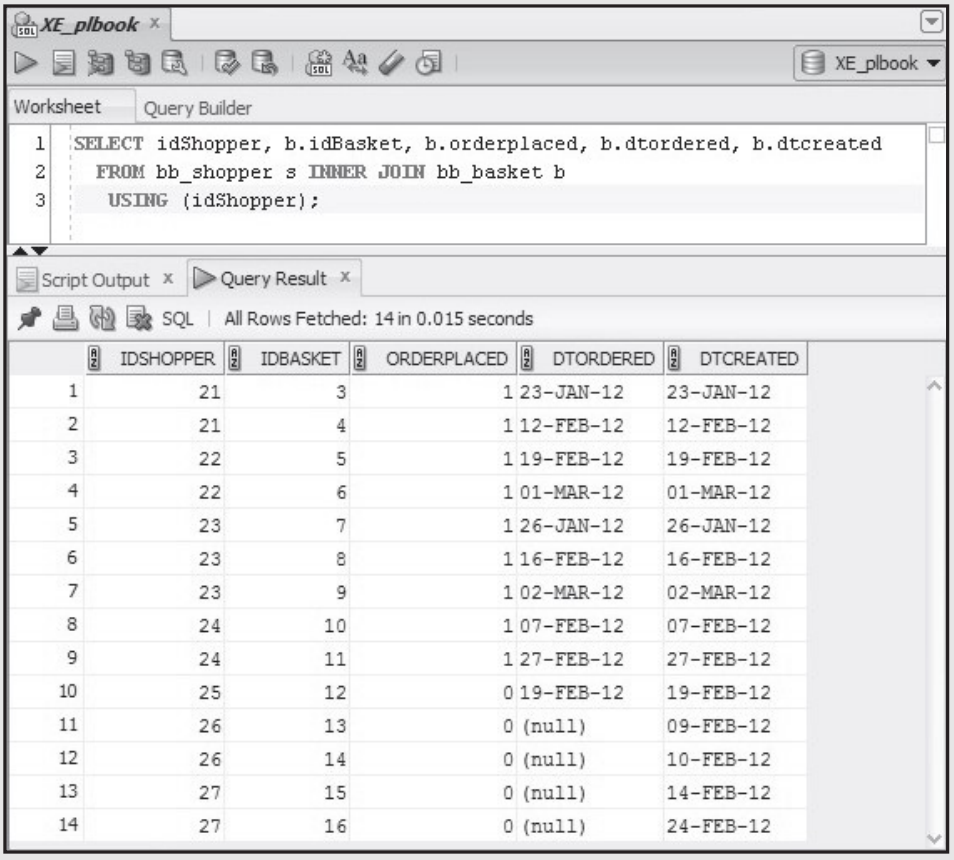
1. Start SQL Developer and connect to Oracle.
2. Enter the query shown in Figure 1-22 and check your data against the listing shown.



	IDPRODUCT	PRODUCTNAME	PRICE	ACTIVE	TYPE	IDDEPARTMENT	STOCK
1	1	CapressoBar Model #351	99.99	1	E	2	23
2	2	Capresso Ultima	129.99	1	E	2	15
3	3	Eileen 4-cup French Press	32.5	1	E	2	30
4	4	Coffee Grinder	28.5	1	E	2	26
5	5	Sumatra	10.5	1	C	1	41
6	6	Guatamala	10	1	C	1	42
7	7	Columbia	10.8	1	C	1	61
8	8	Brazil	10.8	1	C	1	53
9	9	Ethiopia	10	1	C	1	54
10	10	Espresso	10	1	C	1	50

FIGURE 1-22 Querying Brewbean's product information

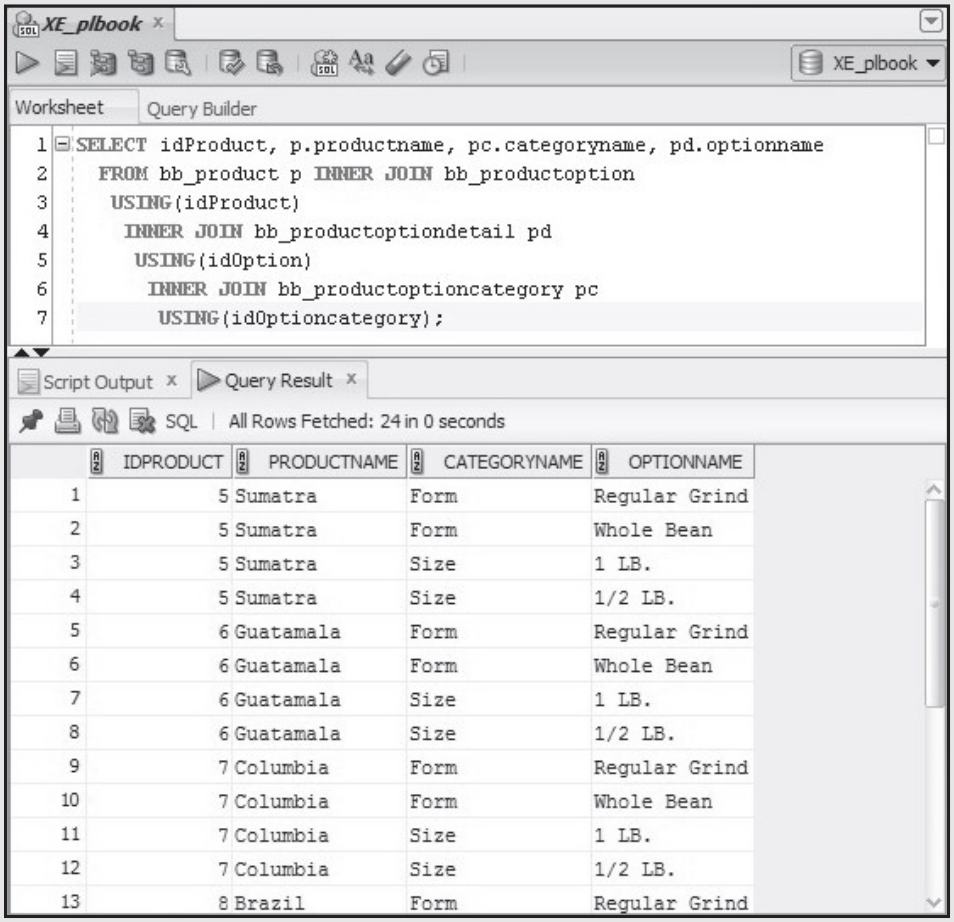
3. Enter the query shown in Figure 1-23 and check your data against the listing shown. Note that ANSI standard joins introduced in Oracle 9*i* are used.



	IDSHOPPER	IDBASKET	ORDERPLACED	DTORDERED	DTCREATED
1	21	3	1	23-JAN-12	23-JAN-12
2	21	4	1	12-FEB-12	12-FEB-12
3	22	5	1	19-FEB-12	19-FEB-12
4	22	6	1	01-MAR-12	01-MAR-12
5	23	7	1	26-JAN-12	26-JAN-12
6	23	8	1	16-FEB-12	16-FEB-12
7	23	9	1	02-MAR-12	02-MAR-12
8	24	10	1	07-FEB-12	07-FEB-12
9	24	11	1	27-FEB-12	27-FEB-12
10	25	12	0	19-FEB-12	19-FEB-12
11	26	13	0	(null)	09-FEB-12
12	26	14	0	(null)	10-FEB-12
13	27	15	0	(null)	14-FEB-12
14	27	16	0	(null)	24-FEB-12

FIGURE 1-23 Querying Brewbean's order information

4. Enter the query shown in Figure 1-24 and check your data for idProduct 5 and idProduct 6 against the listing shown. The full results should contain 24 rows of output.



	IDPRODUCT	PRODUCTNAME	CATEGORYNAME	OPTIONNAME
1	5	Sumatra	Form	Regular Grind
2	5	Sumatra	Form	Whole Bean
3	5	Sumatra	Size	1 LB.
4	5	Sumatra	Size	1/2 LB.
5	6	Guatamala	Form	Regular Grind
6	6	Guatamala	Form	Whole Bean
7	6	Guatamala	Size	1 LB.
8	6	Guatamala	Size	1/2 LB.
9	7	Columbia	Form	Regular Grind
10	7	Columbia	Form	Whole Bean
11	7	Columbia	Size	1 LB.
12	7	Columbia	Size	1/2 LB.
13	8	Brazil	Form	Regular Grind

FIGURE 1-24 Querying Brewbean's product option information

Next, you write and run your own SQL statements:

1. Produce an unduplicated list of all product IDs for all products that have been sold. Sort the list.
2. Show the basket ID, product ID, product name, and description for all items ordered. (Do it two ways—one with an ANSI join and one with a traditional join.)
3. Modify the queries in Step 2 to include the customer's last name.
4. Display all orders (basket ID, shopper ID, and date ordered) placed in February 2012. The date should be displayed in this format: February 12, 2012.
5. Display the total quantity sold by product ID.
6. Modify the query in Step 5 to show only products that have sold less than a quantity of 3.
7. List all active coffee products (product ID, name, and price) for all coffee items priced above the overall average of coffee items.

8. Create a table named CONTACTS that includes the following columns:

Column Name	Data Type	Length	Constraint/Option
Con_id	NUMBER	4	Primary key
Company_name	VARCHAR2	30	Not null
E-mail	VARCHAR2	30	
Last_date	DATE	n/a	Default to current date
Con_cnt	NUMBER	3	Check constraint to ensure that value is greater than 0

9. Add two rows of data to the table, using data values you create. Make sure the default option on the LAST_DATE column is used in the second row added. Also, issue a command to save the data in the table permanently.
10. Issue a command to change the e-mail value for the first row added to the table. Show a query on the table to confirm that the change was completed. Then issue a command to undo the change.

Assignment 1-2: Reviewing Third-Party Software Tools

Table 1-2 lists several third-party PL/SQL software tools. Go to the Web site for one of these tools and describe at least two features that can help in developing PL/SQL code.

Assignment 1-3: Identifying Processing Steps

Review the Brewbean's application page in Figure 1-25. List the logical processing steps that need to occur if the Check Out button is clicked for the next page to display the order subtotal, shipping amount, tax amount, and final total. The shipping costs are stored in a database table by number of items. The tax percentage, stored by state in a database table, is based on the customer's billing state.

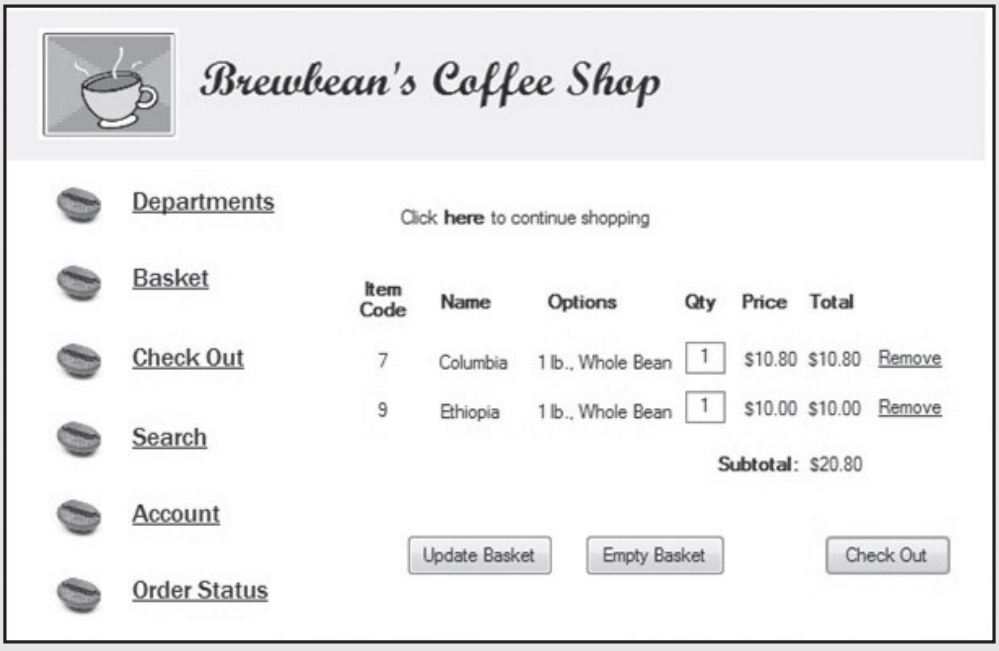


FIGURE 1-25 Brewbean's application page

Assignment 1-4: Using OTN Documentation

Go to the Oracle 11*g* database documentation section on the OTN Web site. Find PL/SQL Language Reference in the list of available resources, and then find the chapter covering control structures. Explain briefly what a control structure is and give an example of a PL/SQL statement that's considered a control structure statement.

Hands-On Assignments Part II

Assignment 1-5: Querying the DoGood Donor Database

Review the DoGood Donor data by writing and running SQL statements to perform the following tasks:

1. List each donor who has made a pledge and indicated a single lump sum payment. Include first name, last name, pledge date, and pledge amount.
2. List each donor who has made a pledge and indicated monthly payments over one year. Include first name, last name, pledge date, and pledge amount. Also, display the monthly payment amount. (Equal monthly payments are made for all pledges paid in monthly payments.)
3. Display an unduplicated list of projects (ID and name) that have pledges committed. Don't display all projects defined; list only those that have pledges assigned.
4. Display the number of pledges made by each donor. Include the donor ID, first name, last name, and number of pledges.
5. Display all pledges made before March 8, 2012. Include all column data from the DD_PLEDGE table.

Case Projects

Case 1-1: Reviewing Oracle SQL Developer Documentation

Locate and review the getting started guide for Oracle SQL Developer on the OTN Web site and describe at least two features you believe will be helpful in creating PL/SQL code.

Case 1-2: Understanding the More Movies Database

In this chapter, you saw the ERD and steps for creating the More Movies database, which is used for the second case project in every chapter. In this project, you perform several queries on the data to become familiar with it and to verify the content. Perform the queries listed in Table 1-3 and compare them with Figures 1-26 to 1-28 to verify your results.

TABLE 1-3 Query tasks for the More Movies database

Query #	What to Query	Results Figure
1	List the ID, title, category ID, and category name for every movie in the database. What five categories are used for movies?	Figure 1-26
2	List the member ID, last name, and suspension code for every member. Are any members suspended at this point?	Figure 1-27
3	List the member last name, rental checkout date, and movie title for all rentals. What checkout data applies to all recorded rentals?	Figure 1-28

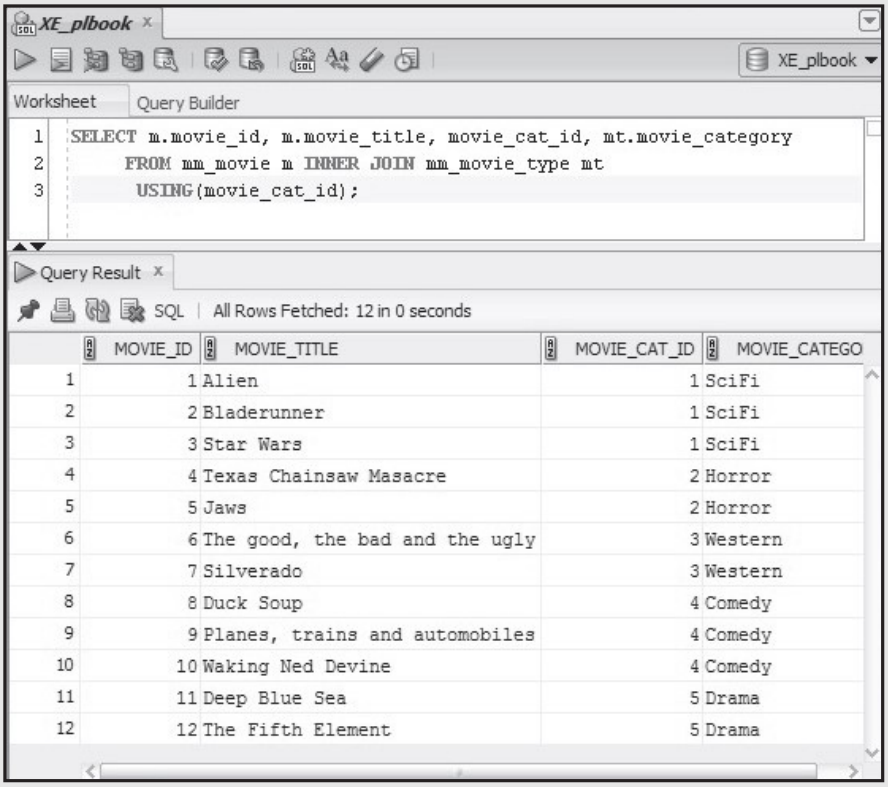


FIGURE 1-26 Results for Query 1

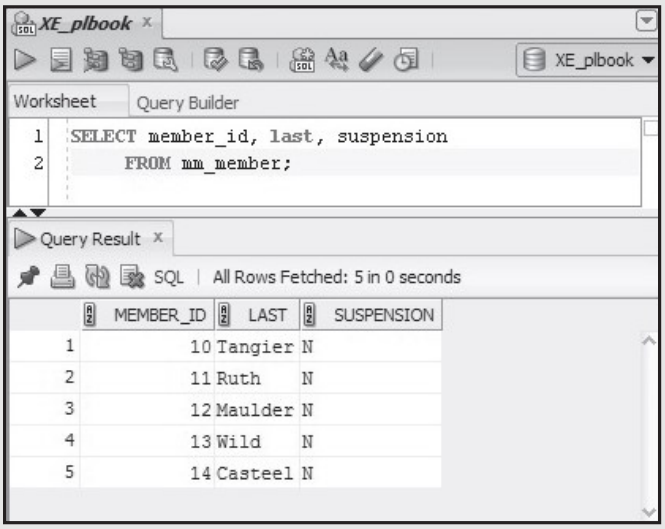


FIGURE 1-27 Results for Query 2

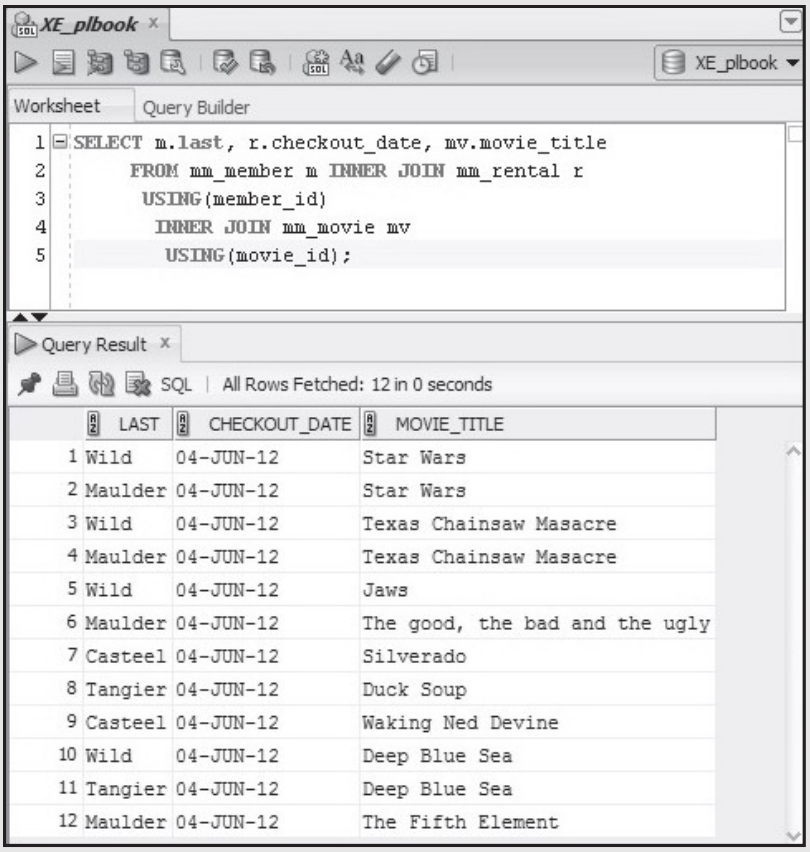


FIGURE 1-28 Results for Query 3

CHAPTER 2

BASIC PL/SQL BLOCK STRUCTURES

LEARNING OBJECTIVES

After completing this chapter, you should be able to understand:

- Programming fundamentals
- PL/SQL blocks
- How to define and declare variables
- How to initialize and manage variable values
- The `NOT NULL` and `CONSTANT` variable options
- How to perform calculations with variables
- The use of SQL single-row functions in PL/SQL statements
- Decision structures: `IF/THEN` and `CASE`
- Looping actions: basic, `FOR`, and `WHILE`
- `CONTINUE` statements
- Nested statements

INTRODUCTION

If you're new to programming, you should learn some basic principles of program design first. A **program** is a set of instructions requesting the computer to perform specific actions, and it might be used to support a system or application. For example, say you're purchasing a book from an online retailer. To begin the purchase, you enter a search for all books containing the term "PL/SQL" in the title. You type this term in a text entry box, and then click a search button. To support this action, a

program is started (triggered) when the search button is clicked. It causes another program to run that reads your text input of "PL/SQL," searches the book database, and displays the results.

A program is also used for activities such as database monitoring. Many database administrators need to monitor system activity, such as the number of active users. A program could be developed to check the number of active users and record the information in the database so that it could be analyzed over time. This program could be scheduled to run automatically at specified periods so the database administrator doesn't need to monitor and record this information manually.

The first section of this chapter gives you an overview of programming fundamentals. Then you begin learning and using the basic PL/SQL block structure.

PROGRAMMING FUNDAMENTALS

Regardless of a program's purpose, all programs have similar basic structures. First, every program has a sequence of actions. Identifying the required actions before coding is essential; therefore, the initial layout of program steps doesn't involve any coding. To illustrate this structure, the Brewbean's application is used. It allows users to enter a term to search for coffee products and then click a search button. Coffee products with the search term in their name or description are then displayed. Figure 2-1 is a flowchart showing a simple layout of the sequence of events that happen after the search button is clicked. Notice that actions are described with text, not code.

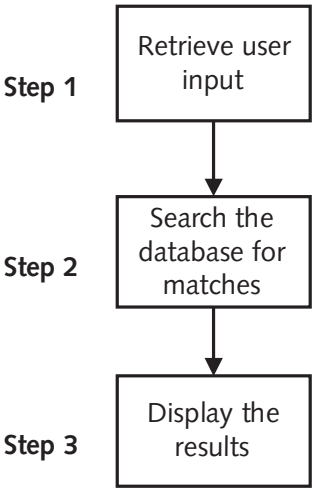


FIGURE 2-1 Flowchart with a sequence structure

NOTE

Some developers prefer to use pseudocode (English-like statements) instead of a flowchart's symbols to describe actions needed for a program.

Second, programs typically include decision structures that outline different sequences of events depending on a question determined at runtime. Take a look at another example involving the Brewbean's application. A customer clicks an order button in the shopping basket screen to place an order online. When this button is clicked, a summary of information, including the total cost, should be displayed. The total cost needs to reflect a 10% discount if the customer is a member of the company's coffee club. Figure 2-2 is a flowchart including a decision structure to determine the sequence of events based on club membership.

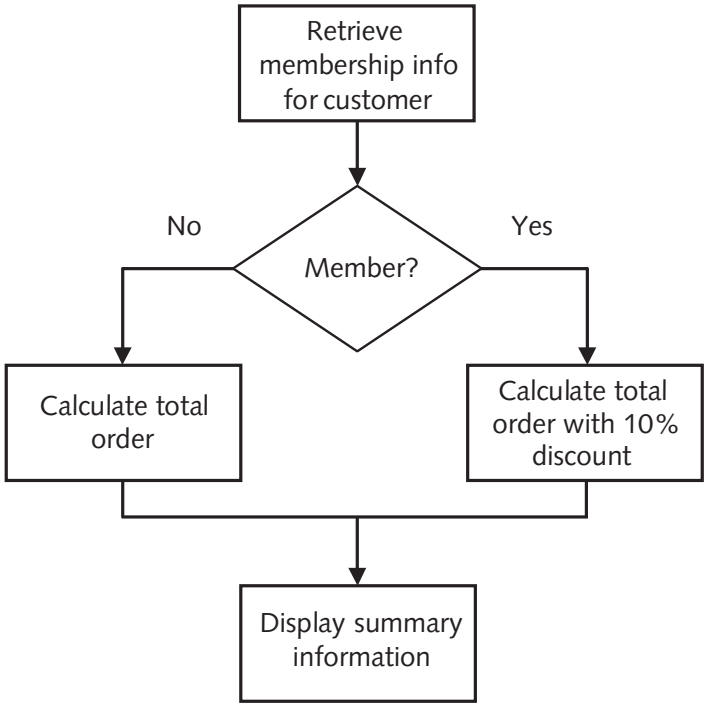


FIGURE 2-2 Flowchart including a decision structure

Third, many programs use a looping structure to repeat actions. For example, Brewbean's might need to calculate new sales prices for all products and store this information in the product database. Say that Brewbean's wants to set all product sale prices to reflect a 10% discount. Figure 2-3's flowchart includes the looping action to repeat the sales price calculation and update for each product in the database. Because the same actions need to occur for each product retrieved from the database, the calculation and update coding can be written once and repeated for each product.

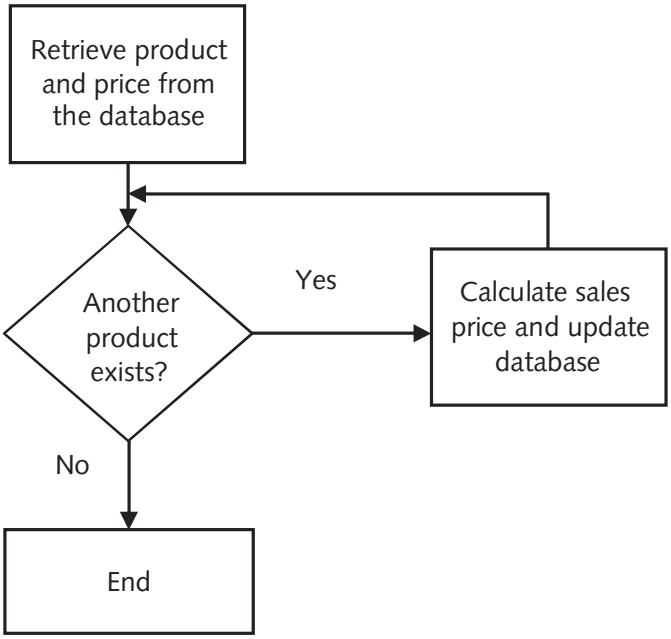


FIGURE 2-3 Flowchart including a looping structure

Now that you have an idea of the basic logic structure involved in programs, you need to translate these actions into PL/SQL code. Chapters 2 through 4 focus on showing PL/SQL coding to address data handling, making decisions, looping, and handling errors. As coding tasks get more complex, you can return to these flowcharting concepts to help you determine the code steps you need for your programs.

The flowcharts shown in this book are basic, using a diamond shape to indicate decision processes and a square shape for all other actions. Formal flowcharting includes a wider variety of symbols and standardized formatting, but this book's intent isn't to teach you formal flowcharting. Flowcharts are included solely as a simple way to organize the processing steps required for a block of code. If you're new to programming, laying out the sequence of processing steps before coding is important. This book uses flowcharts only in the first few chapters to show how they might be used in developing your PL/SQL programs; however, programmers use several other methods to outline program logic flow, including simply writing numbered steps. Don't underestimate the importance of determining program steps before writing code. Even advanced programmers emphasize the importance of this preparation to increase programming productivity and reduce the time spent debugging coding problems.

TIP

Those new to programming should practice using flowcharts or writing coding steps in pseudocode when attempting assignment problems. Determine the method you find most helpful, and use it throughout this book to assist in solving coding problems.

THE CURRENT CHALLENGE IN THE BREWBEAN'S APPLICATION

Before jumping into coding, there's another data-handling challenge in the Brewbean's application you should consider. In the rest of this chapter, you learn PL/SQL coding techniques to address some requirements for this challenge.

The Brewbean's application already has certain functions. As you can see in Figure 2-4, a shopping basket screen shows shoppers all the items and quantities selected so far. A number of processing tasks could be required when the shopper clicks the Check Out button. These tasks include calculating taxes, calculating shipping charges, checking and/or updating product inventory, and determining whether the shopper already has credit card information stored.

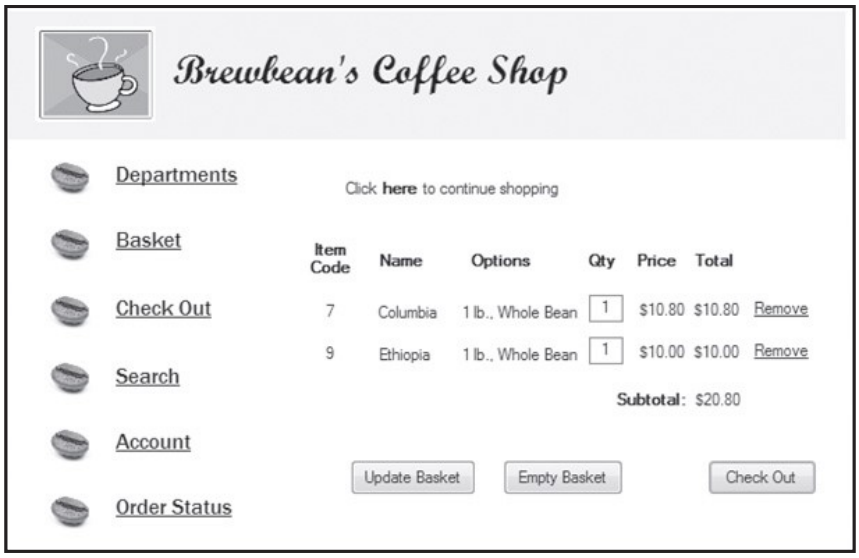


FIGURE 2-4 The Brewbean's shopping basket

So how do you approach programming the required tasks? Typically, a block is written to perform a particular task. For example, for the Brewbean's shopping basket, a PL/SQL block might be written to handle calculating shipping costs based on the number of items in the order and the type of shipping method that's selected. In addition, separate blocks can be developed to calculate applicable taxes and handle inventory issues. Using this modular approach makes developing an application and maintaining and reusing code more manageable than having large, more complex blocks of code.

Keep reusability in mind as you construct blocks or modules of code. Look at the shipping calculation as an example. This block could accept input, such as a basket number, from the shopping basket. The program then might use this input to query the database for the number of items in the basket. By accepting input values, the same PL/SQL block is made dynamic, meaning it can calculate the shipping costs for any order because it queries the database for the total number of items based on the basket number.

PL/SQL BLOCK STRUCTURE

In this section, you start with the basic structure of a PL/SQL block. To learn the syntax of a block, you begin working with **anonymous blocks**, which are blocks of code that aren't stored for reuse and, as far as the Oracle server is concerned, no longer exist after they run. The block code is entered in SQL Developer, just as you would enter an SQL statement.

As outlined in Table 2-1, PL/SQL code is created in a block structure containing four main sections: DECLARE, BEGIN, EXCEPTION, and END. A block is always closed with an `END;` statement. The only required sections of a PL/SQL block are the BEGIN and END sections; however, most blocks use all four sections.

TABLE 2-1 PL/SQL Block Sections

Block Section	Section Content
DECLARE	Creates variables, cursors, and types
BEGIN	Contains SQL statements, conditional logic processing, loops, and assignment statements
EXCEPTION	Contains error handlers
END	Closes the block

The following sections discuss the first three sections in Table 2-1 in detail.

DECLARE Section

The DECLARE section contains code that creates variables, cursors, and data types, used to hold data for processing and manipulation. For example, if a shopper on the Brewbean's Web site clicks the Check Out button, the code needs to retrieve all the item quantities in the shopper's basket to calculate shipping costs. In addition, the code needs to calculate the actual shipping cost. Variables store this data to be used while the block runs.

TIP

As you progress in your studies, you'll discover that a number of different variable types exist, and they serve different purposes. For example, a scalar variable can hold only a single value, whereas a composite variable can hold multiple values.

BEGIN Section

The BEGIN section, the heart of the PL/SQL block, contains all the processing action or programming logic. SQL is used for database queries and data manipulation. Conditional logic, such as `IF` statements (explained later in this chapter), is used to make decisions on what action to take. Loops are used to repeat code, and assignment statements are used to put or change values in variables. In the shipping cost example, you can use `IF` statements to check the quantity of items and apply the correct shipping cost.

EXCEPTION Section

One job of a developer is to anticipate possible errors and include exception handlers to show users understandable messages for corrections that are needed or system problems that must be addressed. The EXCEPTION section contains handlers that enable you to control what the application does if an error occurs during executable statements in the BEGIN section. For example, if your code attempts to retrieve all the items in a basket, but there are no items, an Oracle error occurs. You can use exception handlers so that shoppers don't see Oracle system error messages and the application doesn't halt operation. Instead, you can display easy-to-understand messages to users and allow the application to continue via exception handlers.

NOTE

As you work through the next sections, keep in mind that PL/SQL isn't a case-sensitive language. However, to improve readability, the code examples in this book use uppercase letters for keywords such as `DECLARE` and `BEGIN`.

WORKING WITH VARIABLES

In most PL/SQL blocks, variables are needed to hold values for use in the program logic. **Variables** are named memory areas that hold values so that they can be retrieved and manipulated in programs. For example, if a `SELECT` statement is included in a block, variables are needed to hold data retrieved from the database. In addition, if the block contains a calculation, a variable is needed to hold the resulting value.

The type of data to be stored determines the type of variable needed. For example, if only a single value needs to be stored, scalar variables are used. However, if multiple values need to be stored, such as an entire row from a database table, a record (which is a composite variable) is needed. Finally, if you intend to process a number of rows retrieved with a `SELECT` statement, you might create a cursor, which is a structure specifically suited to processing a group of rows.

Don't feel overwhelmed by these different data types at this point. They're discussed in more detail in subsequent chapters. For now, just remember that a variable is used to hold data, and no matter what type of variable is needed, the variable must be declared in the DECLARE section of the block before you can use it in the BEGIN section.

To declare a variable, you must supply a variable name and data type. Variable names follow the same naming conventions as Oracle database objects:

- Begin with an alpha character
- Contain up to 30 characters
- Can include uppercase and lowercase letters, numbers, and special characters (_, $, and #)
- Exclude PL/SQL reserved words and keywords

TIP

You can find a list of PL/SQL reserved words and keywords in the Oracle PL/SQL User's Guide and Reference, available on the OTN Web site.

Naming conventions are rules used in assigning variable names, and you learn some in the following sections. These rules are helpful to programmers because variable names indicate information about the variable, such as what type of data it stores (a date or a number, for example).

Working with Scalar Variables

Scalar variables, as mentioned, can hold a single value. The common data types for scalar variables include character, numeric, date, and Boolean. Table 2-2 describes them briefly.

TABLE 2-2 Scalar Variable Data Types

Type	Written in PL/SQL Code	Description
Character	`CHAR(n)`	Stores alphanumeric data, with *n* representing the number of characters. The variable always stores *n* number of characters, regardless of the actual length of the value it's storing because it pads the data with blanks. If *n* isn't provided, the length defaults to 1. The maximum length is 32,767 characters. Use `CHAR` for items that always contain the same number of characters. Otherwise, it's more efficient (in terms of system resources) to use `VARCHAR2`.
	`VARCHAR2(l)`	Stores alphanumeric data, with *l* representing the number of characters. The variable stores only the number of characters needed to hold the value placed in this variable, regardless of the actual variable length (hence "VAR" for variable). The *l* value is required. The maximum length is 32,767 characters.
Numeric	`NUMBER(p,s)`	Stores numeric data, with *p* representing the size or precision and *s* representing the scale. The size includes the total number of digits, and the scale is the number of digits to the right of the decimal point. For example, a variable declared as `NUMBER(2,1)` can hold the number 9.9 but is too small for 19.9. If *s* isn't provided, the variable can't store decimal amounts. If both *p* and *s* are omitted, the variable defaults to a size of 40.
Date	`DATE`	Stores date and time values. The default format for Oracle to identify a string value as a date is *DD-MON-YY*. For example, 15-NOV-12 is recognized as a date. This setting is in the `init.ora` file and can be changed by the DBA.
Boolean	`BOOLEAN`	Stores a value of `TRUE`, `FALSE`, or `NULL`. Typically used to indicate the result of checking a condition or group of conditions. In other words, it provides a variable that represents a logical condition's state in terms of true or false. A nonexistent value or `NULL` is more accurately referenced as `UNKNOWN` in terms of a Boolean variable.

Other data types are available that you can use after you're comfortable with basic PL/SQL programming. For example, date data types of `INTERVAL` and `TIMESTAMP` can be used. Also, the numeric data type `SIMPLE_INTEGER`, introduced in Oracle 11*g*, can be useful for arithmetic-heavy operations.

You're already familiar with most of these data types, as they're used as column data types in tables. The Boolean data type might be new to you, however, as it's not a column data type. This data type holds a `TRUE` or `FALSE` value and is used quite a bit with logic that checks for the existence of some condition in a program. For example, a Boolean variable might be set to `TRUE` if the shipping address should be the same as the shopper's address. In your program logic, you can check this flag easily to determine which database fields to use to retrieve the shipping address.

TIP

Many database developers use a CHAR column containing Y/N or a NUMBER column containing 0/1 to mimic a Boolean data type.

In the following sections, you learn how to use scalar variables, starting with their declarations.

Variable Declarations in Code

To give you a picture of what variable declarations look like, the following code snippet displays four scalar variable declarations in the DECLARE section of a PL/SQL block. Notice that only one variable declaration per line is allowed, and each ends with a semicolon.

```
DECLARE
  lv_ord_date DATE;
  lv_last_txt VARCHAR2(25);
  lv_qty_num NUMBER(2);
  lv_shipflag_bln BOOLEAN;
BEGIN
---- PL/SQL executable statements ----
END;
```

Each line of code declares a variable by supplying a variable name and data type, which is the minimum information necessary to declare a variable. The lv_ prefix indicates scalar variables that are local to (created in) the block. The suffixes, such as DATE and NUMBER, indicate the data type for the variable. As you create and modify blocks of code, this information helps you identify the types of values each variable holds. Each variable in the preceding code example contains a NULL value (empty) when the BEGIN section starts execution.

Try the first block, which creates the four variables, assigns a value to each variable, and displays each variable to verify the contents. First, you need to know how to assign values to variables and display variable values. These actions occur in a block's executable or BEGIN section. PL/SQL assignment statements are constructed as follows:

```
variable_name := value or expression;
```

Notice that a colon is used with an equals sign to assign a value. One simple way to display values onscreen in SQL Developer is to use the DBMS_OUTPUT.PUT_LINE procedure. To display the contents or value of a variable named lv_ord_date, you use the following statement:

```
DBMS_OUTPUT.PUT_LINE(lv_ord_date);
```

NOTE

The PUT_LINE procedure is available in the DBMS_OUTPUT Oracle-supplied package.

Be aware that the PUT_LINE procedure displays string values by default and can't display the value for a Boolean variable; if you try to do so, it causes an error. Later in this section, you see an example that includes declaring and assigning a value to a Boolean variable to work around this display limitation.

SQL Developer has an output pane for displaying the output of the PUT_LINE procedure. To prepare for running some code examples, follow these steps:

1. Start SQL Developer and connect to the database.
2. If the Dbms Output pane isn't displayed under the Worksheet, click **View**, **Dbms Output** from the main menu to display it (see Figure 2-5). Notice that the Script Output pane is closed in this figure.
3. Click the green plus symbol on the Dbms Output toolbar to enable this display feature. You need to confirm for which connection this feature is to be activated.

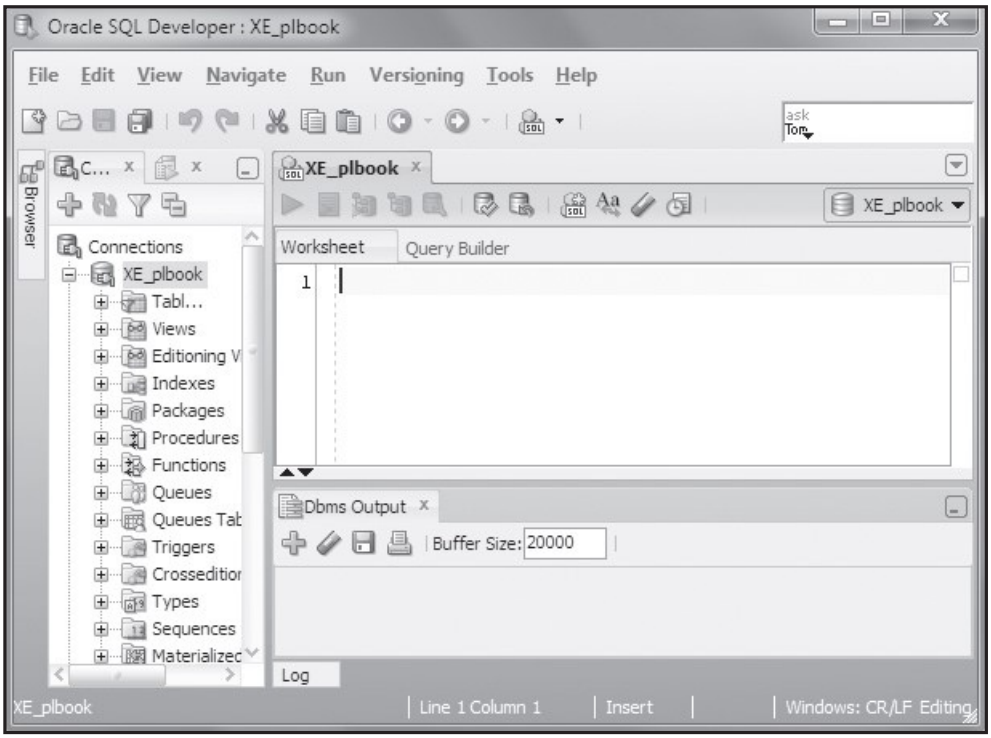


FIGURE 2-5 The SQL Developer Dbms Output pane

NOTE

In subsequent coding examples, it's assumed you have completed the preceding steps.

Now you have all the pieces to build and run the first block. Enter the block of code shown in Figure 2-6 and run it by clicking the Run Script button on the Worksheet toolbar. Verify the execution results in four output lines (see Figure 2-7).

Run Script button

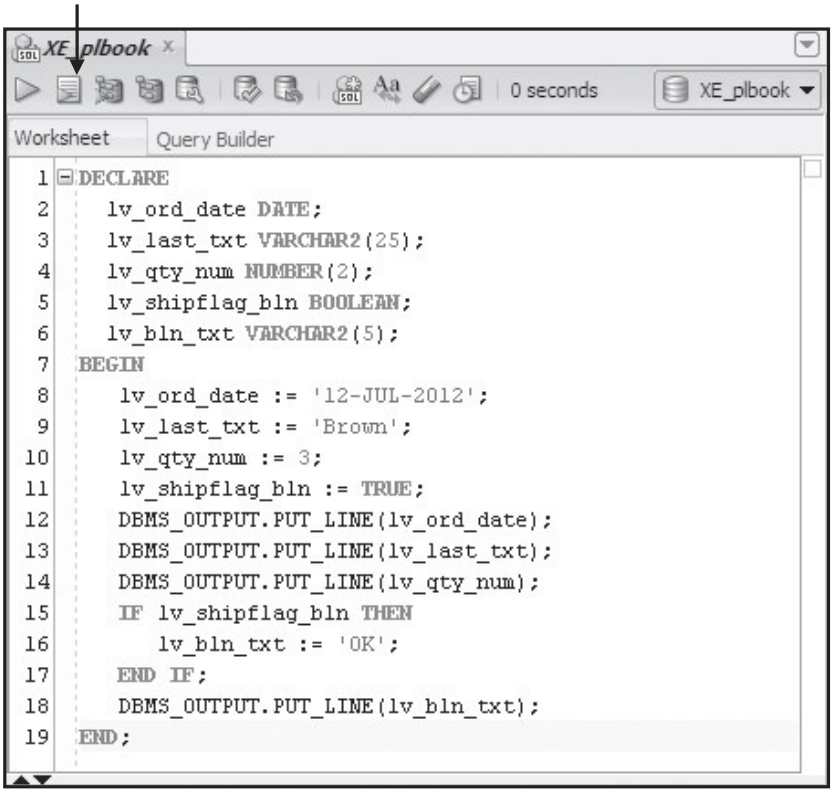


FIGURE 2-6 The block to declare and display variables

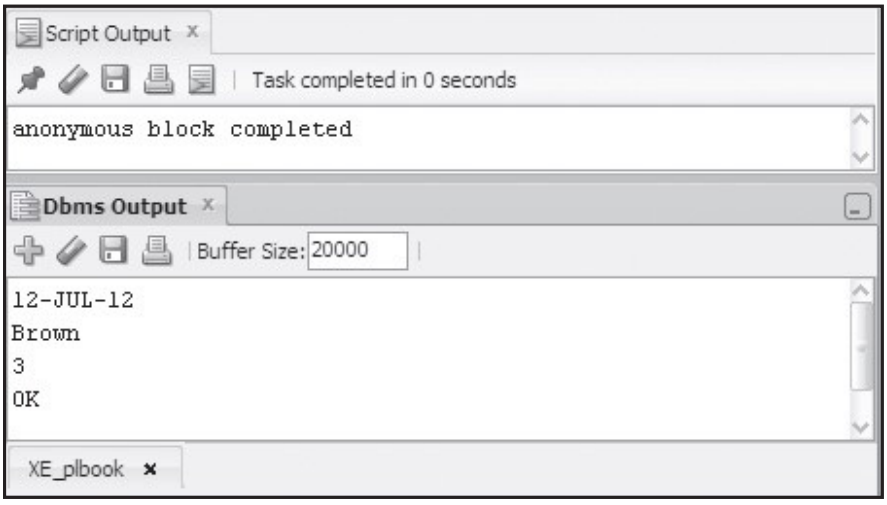


FIGURE 2-7 Results of running the code block

TIP

In the SQL*Plus client tool, a forward slash on the line after the block of code instructs the system to run the code.

PL/SQL programmers should be aware that it's possible to break the rules for variable names, but it's considered a poor practice because it can cause difficulties in reading and debugging code someone else has developed. Using double quotes around an identifier or a variable name is one way to circumvent naming rules, as when you want to begin a variable name with a number or include whitespace in a name. However, doing so makes the variable name case sensitive, which can cause some confusion. Figure 2-8 shows some examples.

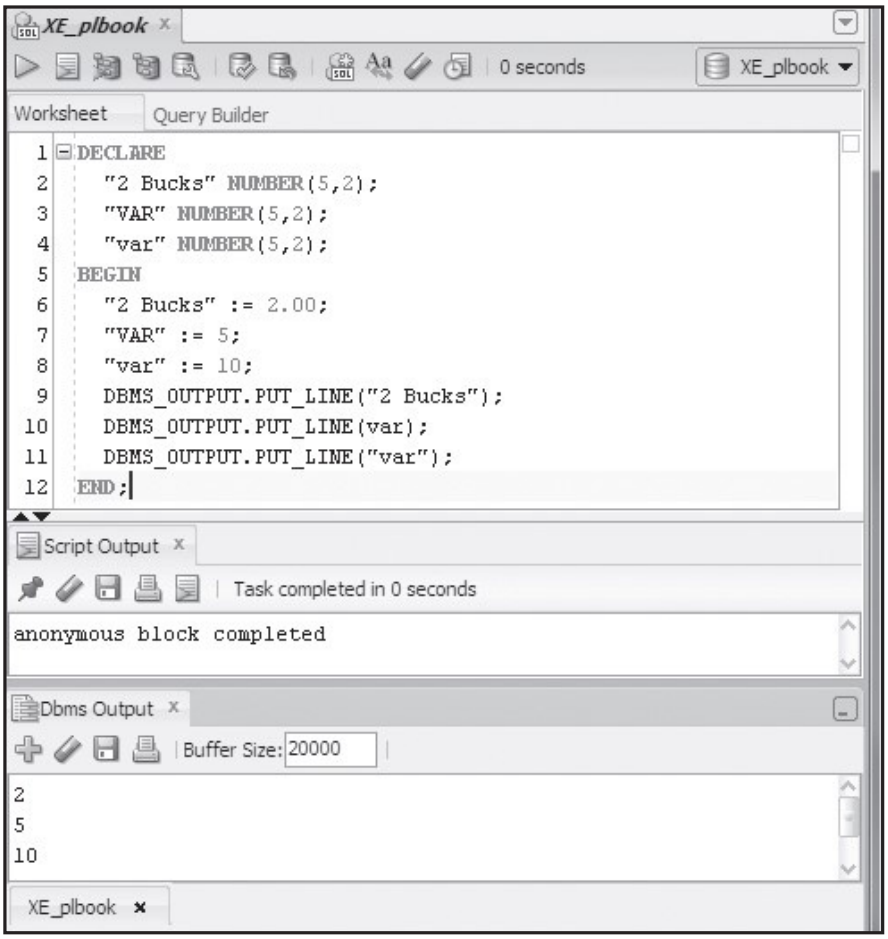


FIGURE 2-8 Using double quotes with variable names

Notice that the `"2 Bucks"` variable name is allowed, even though it starts with a number and includes whitespace. The second line of output returns a 5 for the `var` variable. You might expect a 10 to be returned because the lowercase `var` variable is assigned the value 10. However, the output statement didn't use double quotes around the variable name, so the system defaults to interpreting it as the uppercase `VAR`. The last line of output returns the 10 you expect for `var` because double quotes are used around the variable name. As you can observe with this simple block, using double quotes in variable naming can become confusing quite quickly.

Managing Variable Values

Now that you're familiar with assignment statements and displaying values in PL/SQL, it's time to examine some common hurdles with handling data values. This section describes

common problems in using formatting characters in number values, string conversions on numbers, and including quotes in variable values.

A typical issue with numeric assignment statement is attempting to include formatting characters. For example, when working with monetary amounts, you might accidentally include a dollar sign or commas. Review the code block and error shown in Figure 2-9. An attempt is made to assign the value $5,246.22 to a number variable. Notice that the error references encountering a $ symbol. In SQL Developer, you can detect errors before execution by the color coding in the Worksheet pane; the editor detects a possible issue with the syntax before execution.

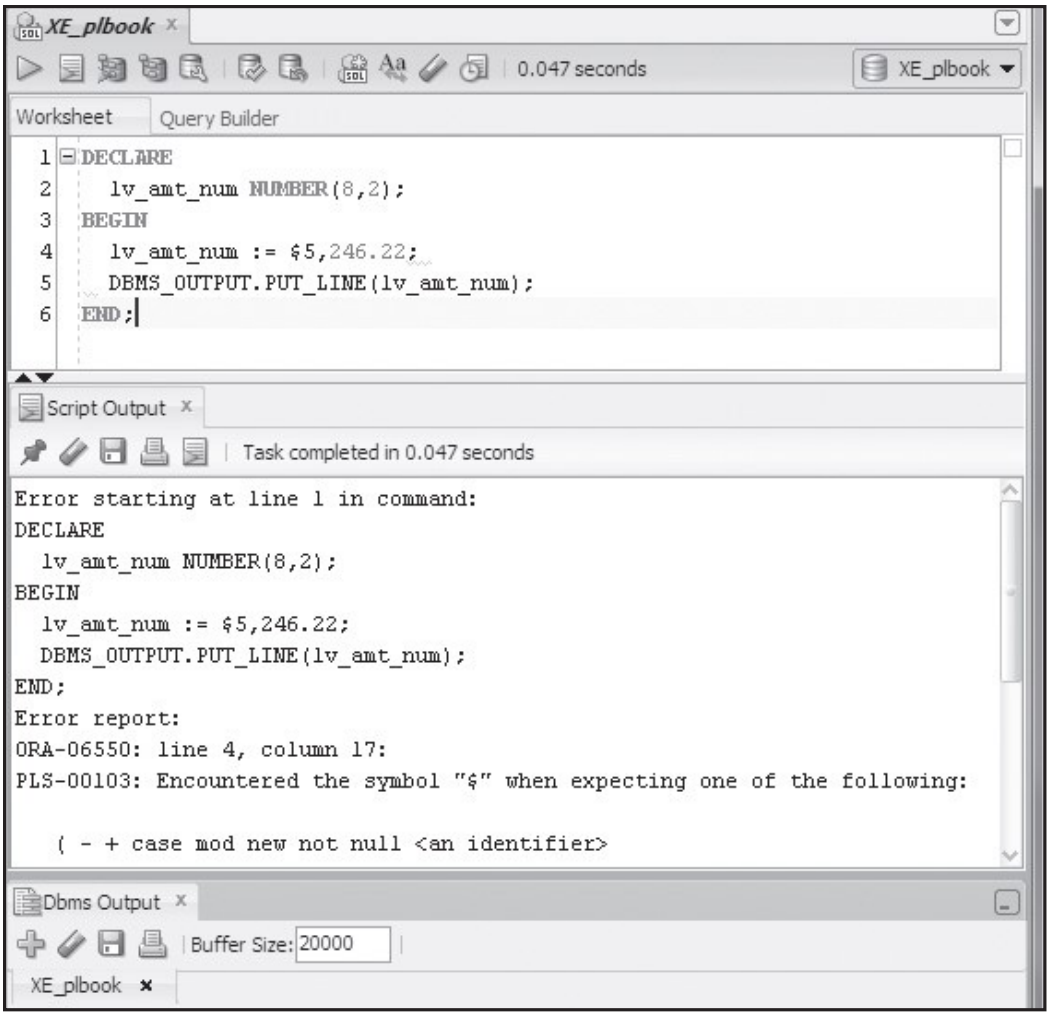


FIGURE 2-9 An error with a number value

String conversion on number values using `TO_CHAR` can also cause problems in rounding and insufficient display length. At times, you need to convert a number value to include formatting for display, such as the dollar symbol and commas shown in the previous example. The `TO_CHAR` function enables you to perform this task; however, you need to be aware of some typical problems associated with format arguments. Review the three output statements and results in Figure 2-10. In the first output, the ####### display indicates that the format argument is too small to hold the value. This format

argument `'$99.99'` allows only two whole numbers, but the value contains three. The second output of $257.85 is displayed accurately because the format argument provides enough length for the value. The third output of $258 is often the most unexpected and can be misleading to users. A format argument with no decimal places automatically rounds a value containing decimal values. Notice that the amount was rounded up to the next whole dollar value.

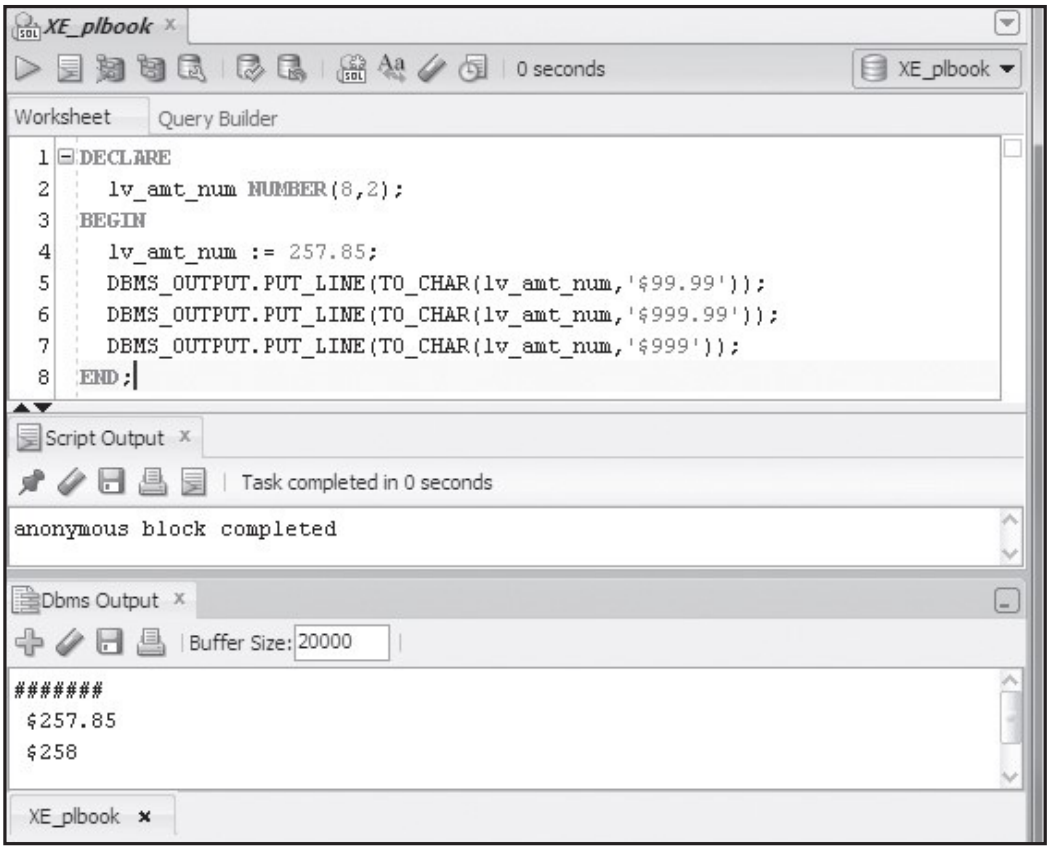


FIGURE 2-10 String conversion issues

Including single quotes in strings can be frustrating because they're used to surround string values in assignment statements. Say you need to assign the string value shown in the following assignment statement. This statement causes an error because it stops at the second single quote to end the string value `Donor` and then can't interpret the remainder of the line: `s Best'`.

```
lv_title_txt := 'Donor's Best';
```

You have two options for including single quotes in string values: Use two single quotes together to represent a single quote in the string or, starting with Oracle 10*g*, use the user-defined delimiters option. Figure 2-11 shows both options. In the first example, note that the string includes two single quotes in succession to represent a single quote in the value; using a double quote doesn't work. The second and third examples use the user-defined delimiter, which begins with the `q` character. Notice the

difference in the two examples. The last one actually includes quotes surrounding the string value as part of the variable value. All text inside the parentheses becomes part of the resulting string value.

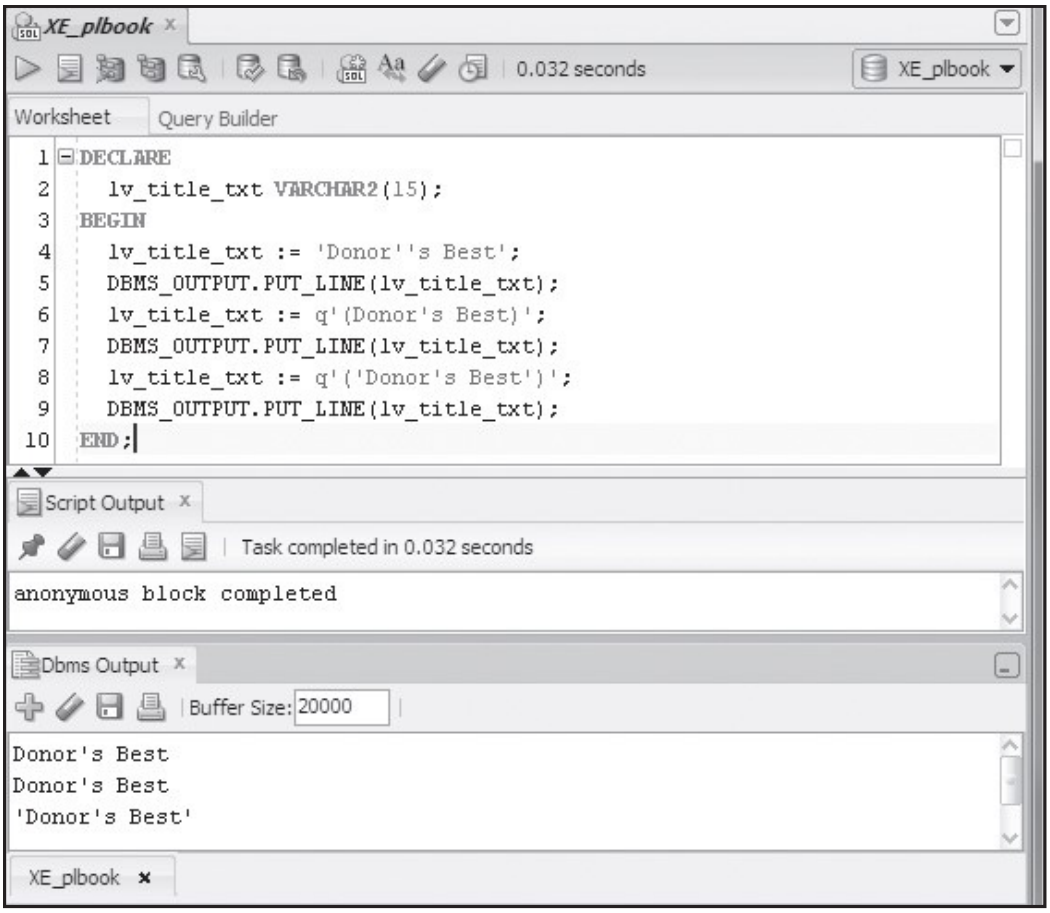


FIGURE 2-11 Including quotes in string values

In the previous examples, the variables don't contain a value until the assignment statements in the BEGIN section run. What if you need a variable to contain a beginning or default value? For example, you might have a date variable that needs to contain the current date. You can set a default value in a variable when it's declared. This technique, called variable initializing, is covered next.

Variable Initialization

Sometimes you need to set a starting value in a variable. You can do this with initialization in the variable declaration so that the variable already contains a value when the block's BEGIN section starts running. To see how to initialize variables in the earlier example, review Figure 2-12 and the changes in the DECLARE section.

The `:=` followed by a value is used to assign initial values to variables in declaration statements. The `DEFAULT` keyword can be used in place of the `:=` symbol to achieve the same result. Typically, a Boolean variable is initialized to `FALSE`, and executable

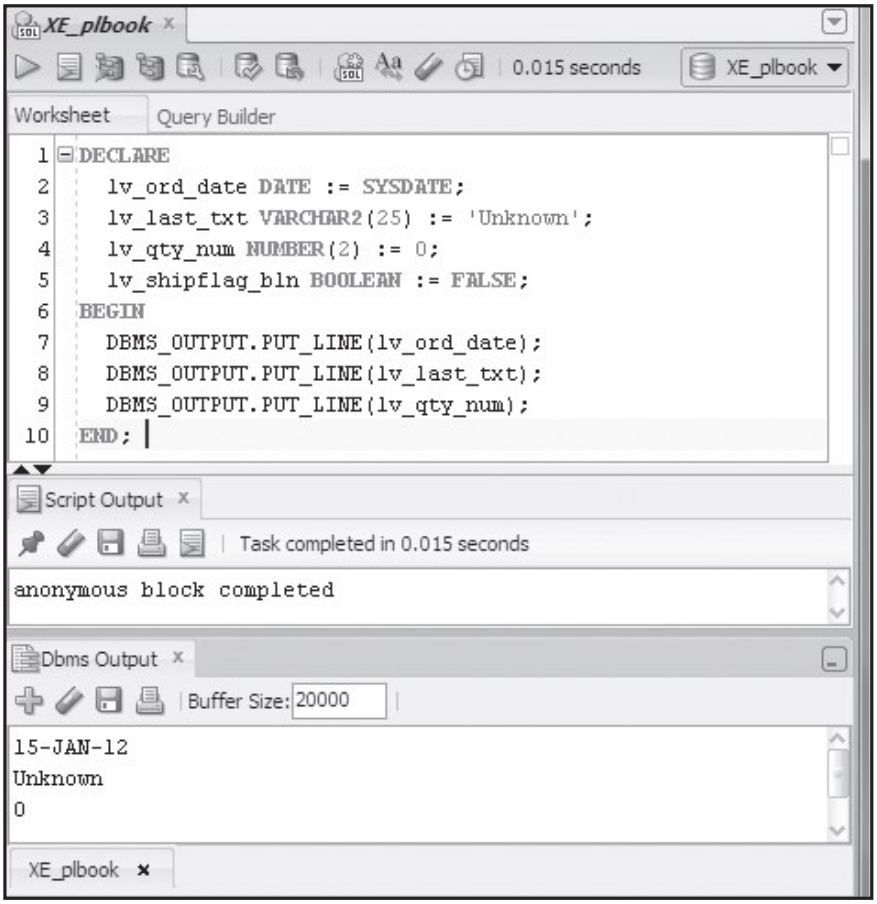


FIGURE 2-12 Variable initialization

statements in the BEGIN section check for conditions and determine whether the variable's value should be changed to TRUE. Numeric variables to be used in calculations are usually initialized to 0 to prevent performing calculations on a NULL value, which isn't the same as a 0 value.

TIP

The DEFAULT keyword can be used in place of the := to get the same result. In practice, the := symbol is more widely used.

Build and run the block in Figure 2-12 with the variable initializations and display all variable values to confirm that the variables contain values as the block begins execution. No assignment statements are performed in the block's executable section. Notice that a display statement isn't included for the Boolean variable. Add the statements needed to confirm that the Boolean variable contains a value from initialization (as explained earlier in "Working with Scalar Variables").

NOT NULL and CONSTANT

In addition to assigning initial values, you can set other controls on variable values with the NOT NULL and CONSTANT options. The NOT NULL option requires that the variable always contains a value. This value can change during the block execution, but the variable must always contain a value. The CONSTANT option can be added

to the variable declaration to ensure that the variable always contains a particular value in the block. That is, it prevents the variable's value from being changed in the block. Take a look at how the NOT NULL and CONSTANT options are included in the variable declaration:

```
DECLARE
  lv_shipcntry_txt VARCHAR2(15) NOT NULL := 'US';
  lv_taxrate_num CONSTANT NUMBER(2,2) := .06;
BEGIN
---- PL/SQL executable statements ----
END;
```

In both declarations, the variable is initialized with a value, using the := assignment symbol. This assignment is required because both options require that the variable always contains a value. In the Brewbean's application, assume the same tax rate applies to all sales; therefore, the lv_taxrate_num variable is declared as a constant to make sure this value isn't modified mistakenly in the block's executable section. The lv_shipcntry_txt variable contains 'US' because at this point, all shoppers are from the United States. Also, note that the CONSTANT keyword is listed before the variable data type, whereas the NOT NULL keywords are listed after the data type.

Try experimenting with these two options in the following steps:

1. Type and run the block shown in Figure 2-13. The errors indicate that both variables require an initialization assignment.

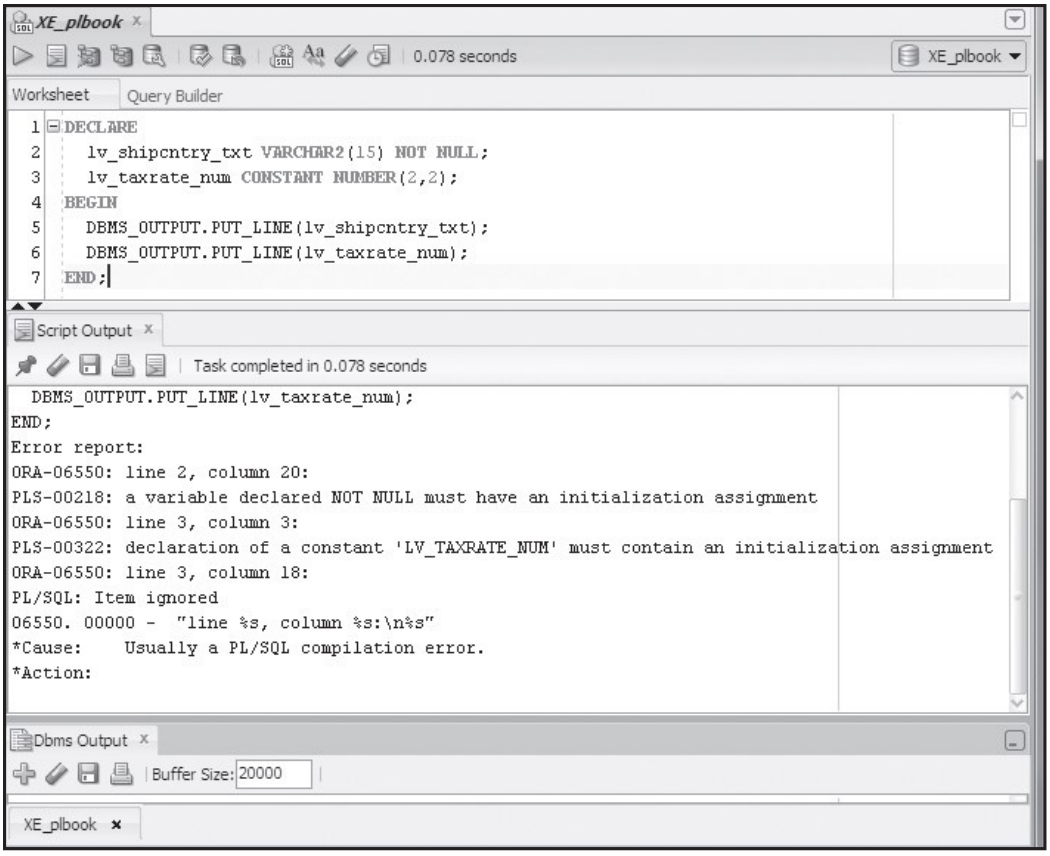


FIGURE 2-13 Error caused by not initializing variables

2. Modify the block to match the following code and run the code. It should run correctly and display the variable values now that the variables are initialized.

```
DECLARE
  lv_shipcntry_txt VARCHAR2(15) NOT NULL := 'US';
  lv_taxrate_num CONSTANT NUMBER(2,2) := .06;
BEGIN
  DBMS_OUTPUT.PUT_LINE(lv_shipcntry_txt);
  DBMS_OUTPUT.PUT_LINE(lv_taxrate_num);
END;
```

3. Modify the block to add the following assignment statement, which changes the tax rate variable's value, and run the code again. An error should be raised stating that the CONSTANT variable can't be used as a target for an assignment statement; it can't be changed in the block.

```
lv_taxrate_num := .08;
```

Calculations with Scalar Variables

You can perform calculations with scalar variables. The following example includes a calculation to determine the tax amount for an order:

```
DECLARE
  lv_taxrate_num CONSTANT NUMBER(2,2) := .06;
  lv_total_num NUMBER(6,2) := 50;
  lv_taxamt_num NUMBER(4,2);
BEGIN
  lv_taxamt_num := lv_total_num * lv_taxrate_num;
  DBMS_OUTPUT.PUT_LINE(lv_taxamt_num);
END;
```

The BEGIN section in the preceding block contains one assignment statement that involves a calculation. All variables declared are used in this statement. The multiplication needed is 50 * .06, or 3, so the `lv_taxamt_num` variable holds this value. Try creating and running this block. It should display the value 3, which is contained in the `lv_taxamt_num` variable.

Again, notice that the `:=` symbol (beginning with a colon) is used to create the assignment statement. Many developers become accustomed to using an equals sign in SQL and forget the colon needed in PL/SQL—don't be one of them! Remove the colon from the assignment statement and run the block again to review the error, shown in Figure 2-14.

Keep in mind that you have hard-coded values for the first two variables to indicate a tax rate and an order total. As you advance in learning PL/SQL coding, you'll see that values such as the order total are typically supplied by the application and could be different for each shopper. For now, you'll continue hard-coding (assigning) values to variables for testing.

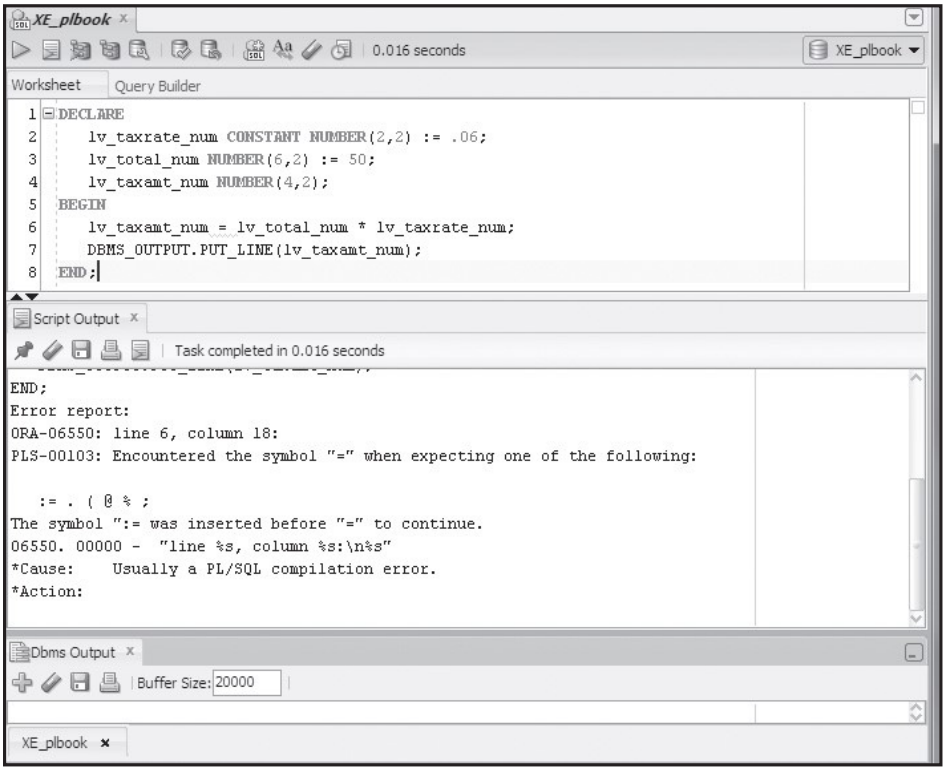


FIGURE 2-14 Error caused by a missing colon in the assignment operator

TIP

The Oracle error messages in the previous examples display both an ORA error number and a PLS error number. You should be familiar with ORA errors from your SQL experience. PLS errors are specific to PL/SQL and can be looked up in Oracle documentation or with an Internet search. Keep in mind that specific errors can involve many different coding problems, so some messages might not seem clear. Always start your investigation on both the coding line referenced in the error and the line before.

Because a number of concepts involving variables have now been covered, try a couple of blocks on your own as practice. Challenge problems, such as those following this section, are sprinkled throughout chapters to give you the opportunity to apply the concepts you're learning. You can find solutions for challenges in this book's student data files. Try these problems on your own before reviewing the solutions to help ensure that you have learned and can apply the skills. Keep in mind that you might have variable names that are different from those in the solutions, which is fine. For these challenges, compare the number of variables, the data types assigned, use of initialization, options applied, and assignment statements in the block's executable section.

CHALLENGE 2-1

Create a block containing a variable that can hold a date value (for a billing date), a variable that can hold a character string (for a last name), and a variable that can hold a numeric value (for a credit balance). For variable names, follow the naming conventions introduced in this chapter. Initialize the date variable to the value October 21, 2012 and the numeric variable to 1,000. In the executable (BEGIN) section of the block, assign the character variable's value as Brown, and include statements to display the value of each variable.

CHALLENGE 2-2

Create a block containing a variable that can hold a Y or an N to indicate whether a newsletter should be mailed, a variable that can hold a balance owed for a customer (initialize to $1,200), a variable that can hold a two-digit decimal value representing the minimum payment rate (initialize to .05), and a variable to hold the minimum payment amount to be calculated. Make sure the value in the variable for the minimum payment rate can't be modified in the block. The newsletter variable should always contain a value, and the default value should be set to Y. In the executable section of the block, calculate the minimum balance owed (balance times the minimum payment rate), set the newsletter variable to N, and display the value of each variable. After you have finished these tasks, add a statement at the top of the executable section to assign a value of .07 to the minimum payment rate. When you run the code, this statement produces an error. Why?

Using SQL Functions with Scalar Variables

SQL single-row functions can be used in PL/SQL statements to manipulate data. You might recall the `ROUND` function, the `SUBSTR` function, and the `UPPER` function, for example. These functions can provide a variety of data manipulation tasks with scalar variables. For example, say you have a block containing two date variables, and you want to calculate the number of months between these two dates. Figure 2-15 shows a block that performs this calculation and displays the result, using the dates September 20, 2010, and October 20, 2012.

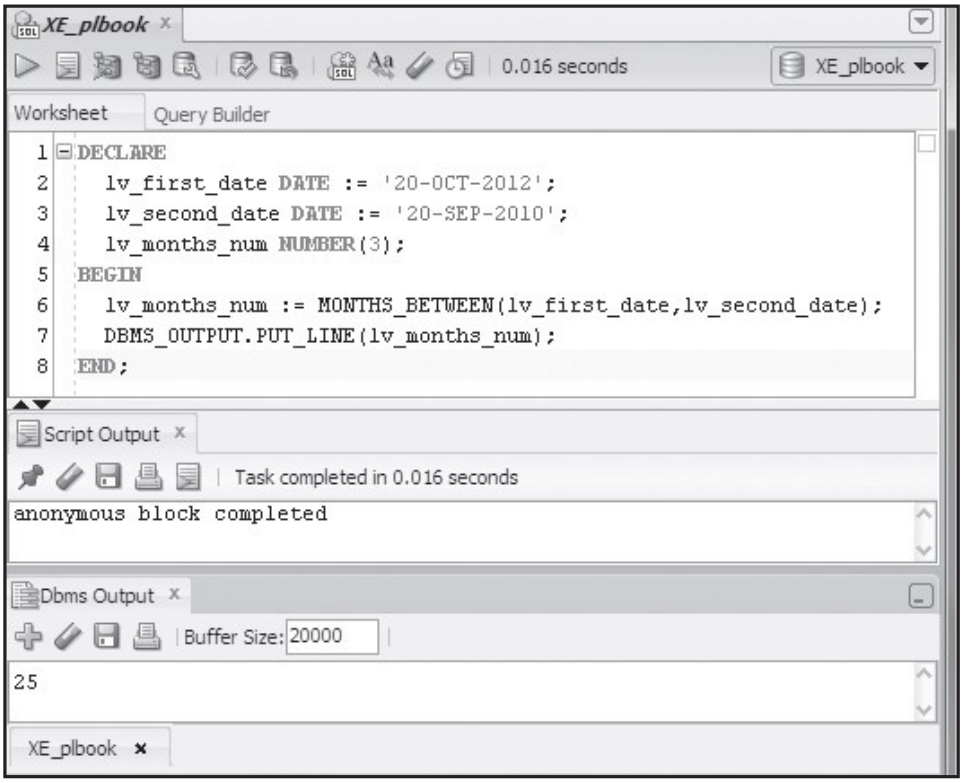


FIGURE 2-15 Using single-row functions in an assignment statement

Notice that two date variables are declared and initialized with the date values indicated. Then a numeric variable is declared to hold the number of months between the two dates. The assignment statement in the executable section uses the `MONTHS_BETWEEN` function to determine the difference in months. The result of the function is held in the

lv_months_num variable. This variable is declared with a number data type because the MONTHS_BETWEEN function returns a numeric value.

CHALLENGE 2-3

Create a block containing a variable for a promotion code that's initialized to the value A0807X. The second and third characters of the promotion code value indicate the month the promotion is applied. The fourth and fifth characters of the promotion code indicate the year the promotion is applied. The code in the block needs to extract these characters. Extract the month and year values separately, and display the values.

Now that you have declared and used scalar variables, you need to look at statements for performing decision making and looping. The statements used to control the flow of logic processing in programs are commonly called **control structures**. They provide the capability to use conditional logic that determines which statements should run, how many times statements should run, and the overall sequence of events. The PL/SQL coding techniques for these tasks are covered next.

DECISION STRUCTURES

Decision structures determine which statements in a block are executed. For example, if the tax amount is assigned based on the state the customer lives in, the code needs to assign a different tax rate based on the state value. There are different types of decision control structures; the following sections cover the condition-processing capabilities of IF-THEN and CASE statements.

Simple IF Statements

An IF statement is a mechanism for checking a condition to determine whether statements should or shouldn't be processed. These statements are often used to check values of variables and direct the processing flow to the particular statements that need to be processed based on the values checked. Review the syntax of an IF statement:

```
IF condition THEN statement END IF;
```

The IF portion of the statement lists the condition to check, and the THEN portion specifies the action to take if the IF portion is true.

Take a look at IF statements in the context of the Brewbean's application. The PL/SQL block that calculates taxes for online orders might need to determine whether the order is from a state requiring taxes. This situation starts with the following assumptions:

- Taxes should be applied only in the company's home state of Virginia.
- You need to check for only one condition: whether the shipping state is Virginia. If it is, the tax amount is calculated by using the order subtotal.
- If the shipping state isn't Virginia, no calculation is performed. The tax amount is then 0.

When an IF statement checks only one condition and performs actions only if the condition is TRUE, it's referred to as a simple IF condition. In the example, the condition that needs to be checked is whether the state is Virginia. The action needed if the condition is true is performing a calculation to determine the tax amount. The tax amount should be set to 0 if the condition is FALSE.

The block in Figure 2-16 includes variables for the state and subtotal values. Notice that the state is set to 'VA'; therefore, the condition in the IF statement is true, and the tax amount is calculated by using 6%. What if the shipping state isn't VA? The IF condition resolves to FALSE, and the tax calculation doesn't take place.

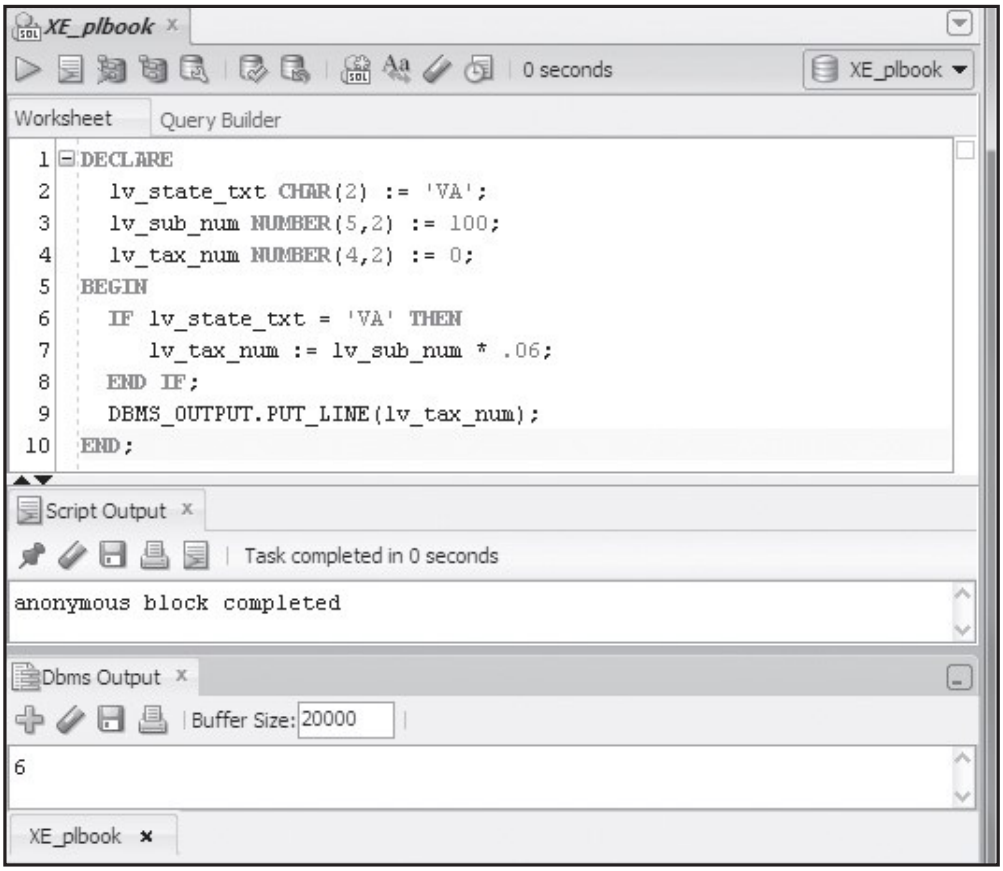


FIGURE 2-16 A basic IF statement

TIP

The = symbol is used in the IF statement to check for a value of VA. It's not an assignment statement, so the := symbol isn't used.

As an application developer, your testing must include a variety of data situations, so try testing the same block, using a shipping state of NC (North Carolina). Modify the block in Figure 2-16 to initialize the state variable to NC, and run the code. The output should confirm that the tax amount is 0 because the shipping state isn't VA, and the tax amount variable is initialized to 0. In this case, the IF condition resolves to FALSE, so the lv_tax_num assignment statement isn't processed. If the tax amount variable hadn't been initialized to 0, it would contain a NULL value.

IF/THEN/ELSE Statements

A simple IF statement performs an action only if the condition is TRUE. What if you need to perform one action if the condition is TRUE and a different action if the condition is FALSE? Say that if the shipping state is VA, a tax rate of 6% is applied, and if the shipping state is anything other than VA, a tax rate of 4% is applied. An ELSE clause is added to

indicate actions to perform if the condition is FALSE. The following example shows the syntax for this statement:

```
IF condition THEN statement
ELSE statement
END IF;
```

CHALLENGE 2-4

Because blocks are beginning to include more execution steps, return to the flowcharts introduced at the beginning of the chapter to help you lay out the sequence of actions needed before you begin coding. Create a flowchart for the tax rate calculation covered in this section.

The block in Figure 2-17 reflects adding an ELSE clause to the IF statement, which uses a 6% tax if the condition checking the shipping state resolves to TRUE. If the condition resolves to FALSE, a 4% tax rate is used. In this example, the state is NC; therefore, a tax rate of 4% is applied. Note that by using an IF/THEN/ELSE statement, one tax amount calculation always occurs, regardless of the shipping state's value. Build the block and test with different state values.

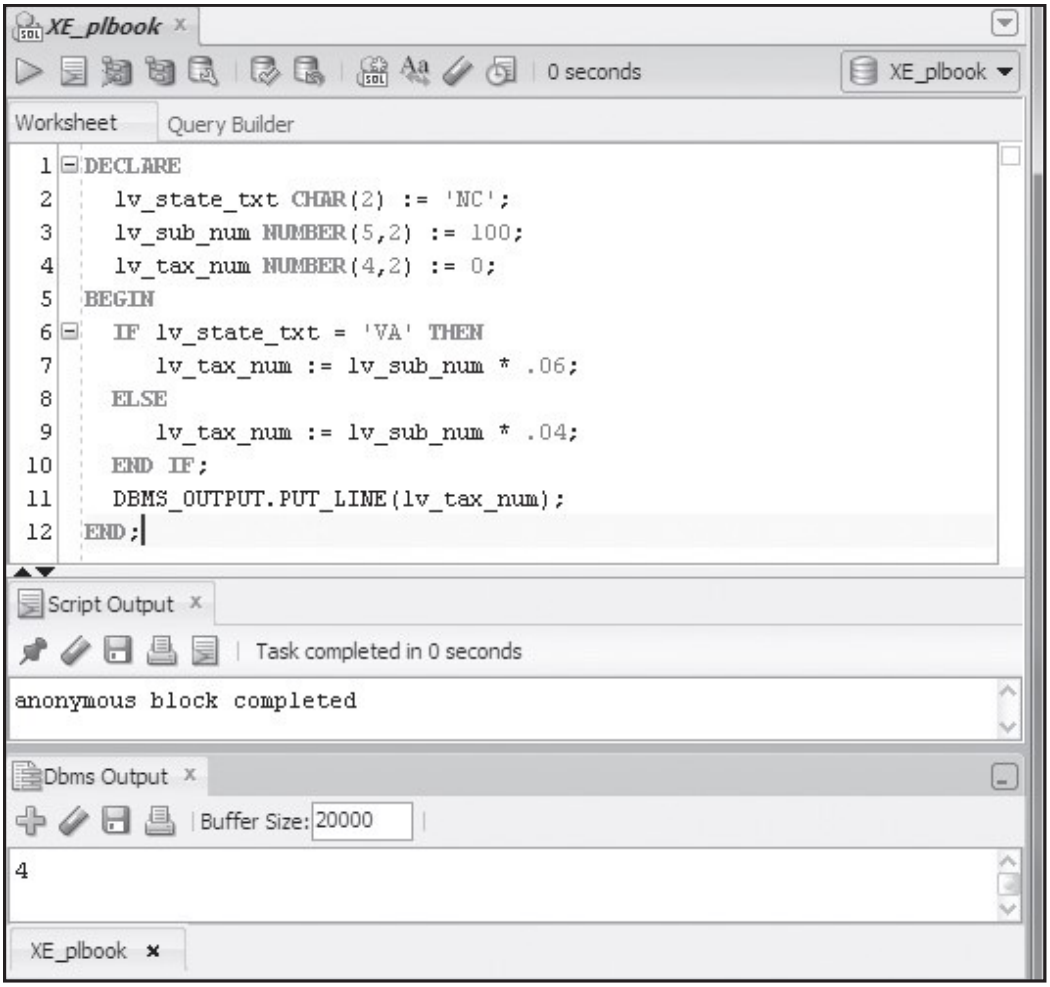


FIGURE 2-17 Adding an ELSE clause to an IF statement

IF/THEN Versus IF

Instead of using the ELSE clause, what if you create the IF tax condition with two simple IF statements, as shown in the following code?

```
IF lv_state_txt = 'VA' THEN
  lv_tax_num := lv_sub_num * .06;
END IF;
IF lv_state_txt <> 'VA' THEN
  lv_tax_num := lv_sub_num * .04;
END IF;
```

This code produces the same result as the IF/THEN/ELSE statement. Which method should be used? One method operates more efficiently than the other. In the preceding code, each IF clause is processed regardless of the shipping state value. On the other hand, the ELSE clause in Figure 2-17 processes only one IF clause (checking the value of the shipping state). This is an important difference in processing efficiency: The less code to process, the faster the program runs.

IF/THEN/ELSIF/ELSE

Now go a step further with this tax calculation example. What if it changes so that a tax rate of 6% should be applied to VA, 5% to ME, 7% to NY, and 4% to all other states? Then you need to check for the existence of several different state values because you no longer have an either/or situation. An ELSIF clause is added to handle checking a variety of conditions for the state value. The syntax for this statement is as follows:

```
IF condition THEN statement
ELSIF condition THEN statement
ELSE statement
END IF;
```

CHALLENGE 2-5

Create a flowchart for the condition described in this section.

Next, convert the flowchart to code, using the IF/THEN/ELSIF/ELSE syntax. Review the code in Figure 2-18, which checks for the existence of different values by using ELSIF clauses. Note that each condition in the ELSIF clauses is mutually exclusive, meaning that only one of the ELSIF conditions can evaluate to TRUE.

The processing begins at the top of the IF statement by checking the shipping state value until it finds a condition that resolves to TRUE. After a TRUE condition is found, the associated program statements are processed, and the IF statement is finished. The program then runs the statement immediately after the END IF; line. If no condition resolves to TRUE, the program statements in the ELSE clause are processed. An ELSE clause isn't required, and if none had been provided

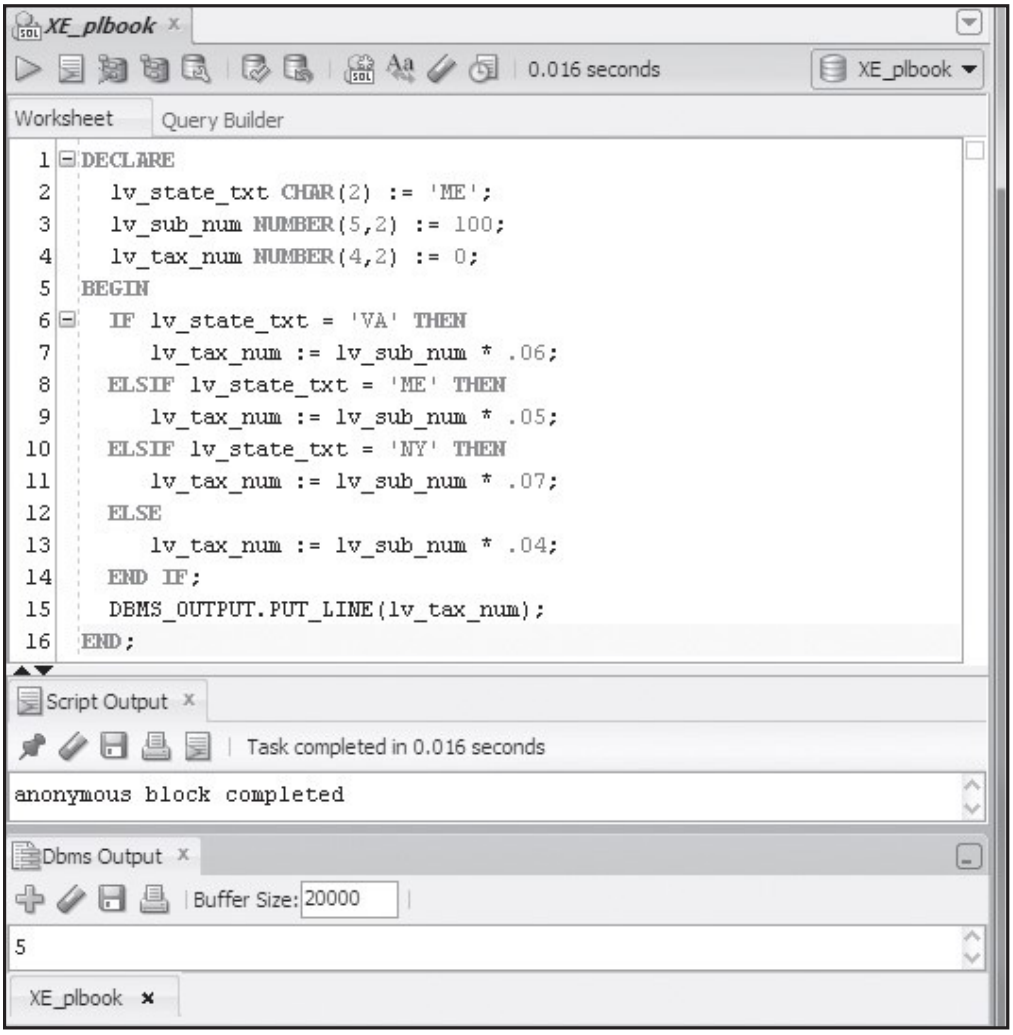


FIGURE 2-18 Adding more conditions with `ELSIF` clauses

in this example, the `IF` statement could be completed without processing a tax calculation.

NOTE

Because `IF` clauses are evaluated from the top down, knowing the nature of your data can make the code more efficient. If much of the data processed matches a particular value, for example, list this value in the first `IF` condition so that only one `IF` clause typically has to run.

You might be tempted to put an "E" in the `ELSIF` keyword, but this is one of the most common mistakes PL/SQL programmers make. Figure 2-19 shows the error caused by using `ELSEIF` rather than `ELSIF`. The error is raised on the term that follows the `ELSEIF`. Because this spelling doesn't represent a keyword, the system thinks the `ELSEIF` is something else, such as a variable.

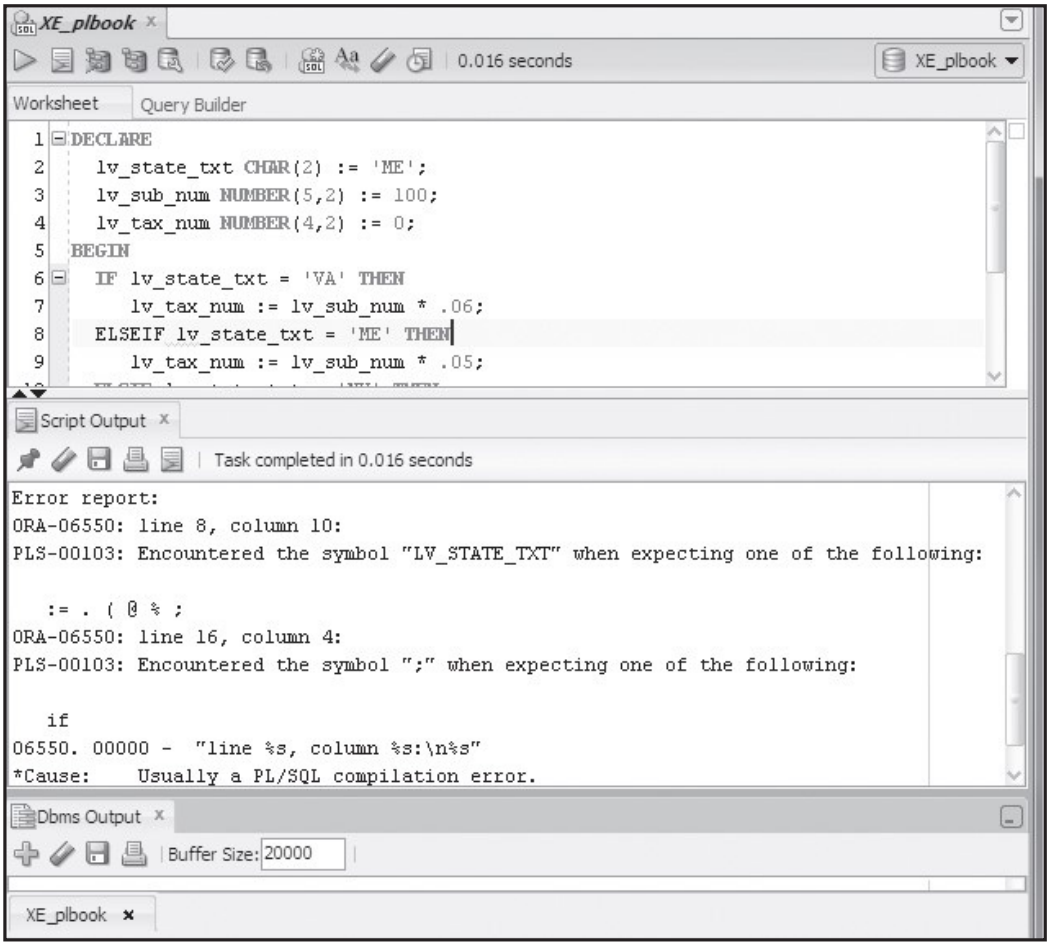


FIGURE 2-19 Error caused by misspelling `ELSIF`

`IF` Statement Evaluations

Now that you understand the `IF` statement structure, it's time to look at factors involved in `IF` statement evaluations, such as `NULL` values, Boolean testing, and order of conditions. When writing an `IF` statement, you need to determine how a `NULL` value in conditions should be handled. For example, Brewbean's needs a block of code to display "Whole Bean" (option value 3) or "Ground" (option value 4) based on the option selected on a coffee product order. Review the following code for performing this task:

```
DECLARE
  lv_option_num NUMBER(2) := 3;
BEGIN
  IF lv_option_num = 3 THEN
    DBMS_OUTPUT.PUT_LINE('Whole Bean');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Ground');
  END IF;
END;
```

What results if the option variable is empty or NULL? Figure 2-20 confirms that the ELSE clause runs if the option variable value is anything except 3, including NULL. This result might not be what you intend, however.

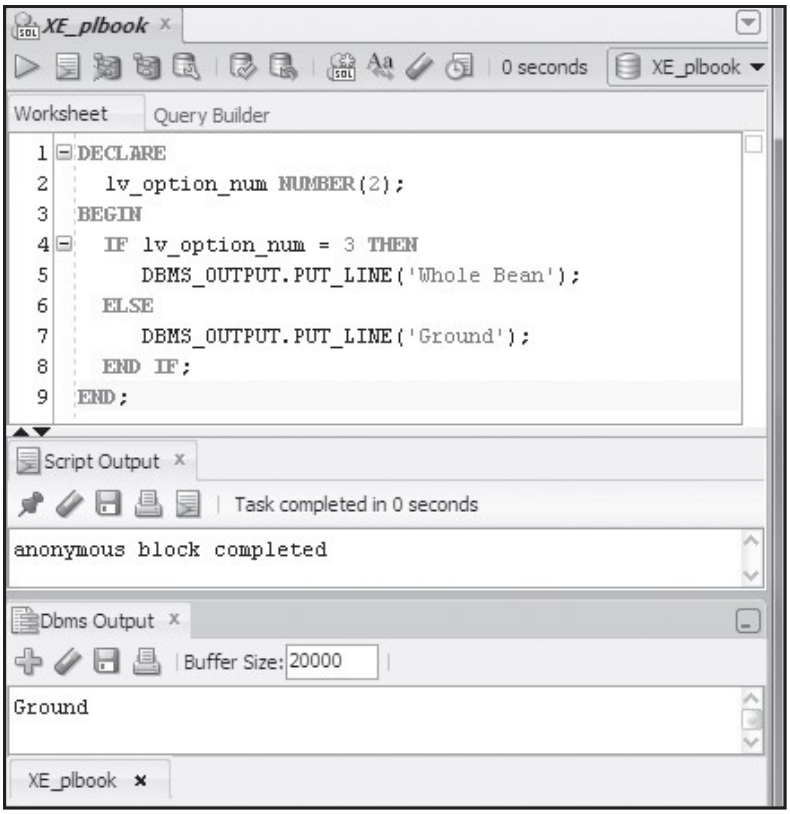


FIGURE 2-20 NULL values with IF conditions

If you know the option variable has possible values of 3, 4, and NULL, and you want to display N/A if a NULL value is detected, you could edit the IF statement several different ways. You can address each possible value with an IF condition and allow the ELSE clause to handle the NULL values, as shown:

```
IF lv_option_num = 3 THEN
  DBMS_OUTPUT.PUT_LINE('Whole Bean');
ELSIF lv_option_num = 4 THEN
  DBMS_OUTPUT.PUT_LINE('Ground');
ELSE
  DBMS_OUTPUT.PUT_LINE('N/A');
END IF;
```

Another alternative is addressing all possible values with IF conditions, including the NULL value, by using the IS NULL comparison operation shown in the following example. The ELSE clause is added to handle any unexpected values for the option selection.

```
IF lv_option_num = 3 THEN
  DBMS_OUTPUT.PUT_LINE('Whole Bean');
ELSIF lv_option_num = 4 THEN
  DBMS_OUTPUT.PUT_LINE('Ground');
ELSIF lv_option_num IS NULL THEN
  DBMS_OUTPUT.PUT_LINE('N/A');
ELSE
  DBMS_OUTPUT.PUT_LINE('Unknown');
END IF;
```

Recall that the functions to check for or convert NULL values for easier handling are IS NULL, IS NOT NULL, NVL, and NVL2.

In terms of Boolean values, an IF statement can be used to determine the value of a Boolean variable; however, it might be possible to eliminate the IF statement and simply use an expression to set the Boolean value. For example, Brewbean's needs a block of code using a Boolean variable to indicate whether an order is considered a delayed shipment. Any order taking more than three days to ship is considered a late shipment. The following block accomplishes this task by using an IF statement to set a Boolean variable value:

```
DECLARE
  lv_ord_date DATE := '15-SEP-2012';
  lv_ship_date DATE := '19-SEP-2012';
  lv_shipflag_bln BOOLEAN;
BEGIN
  IF (lv_ship_date - lv_ord_date) > 3 THEN
    lv_shipflag_bln := TRUE;
  ELSE
    lv_shipflag_bln := FALSE;
  END IF;
  --- Additional PL/SQL statements ---
END;
```

Even though the preceding code works fine, there's a better alternative. A Boolean value can be set with an assignment statement, which requires evaluating an expression. The following statement replaces the entire IF statement and sets the value of the Boolean variable:

```
lv_shipflag_bln := (lv_ship_date - lv_ord_date) > 3;
```

Last, because IF statements always evaluate from top to bottom and execute the first TRUE clause encountered, considering the order of IF conditions is critical. Review the block of code and execution results shown in Figure 2-21. Even though the lv_amt_num variable value is 30, the block executed the first IF clause rather than the second because 30 is greater than 20, resulting in a TRUE condition. This seems obvious with a short statement, as in this example, but it can be overlooked easily when statements become more complex. An IF statement should be tested by invoking every conditional clause to ensure correct operation. Notice that the block example in Figure 2-21 embeds an output statement in each clause to verify which clause is executed.

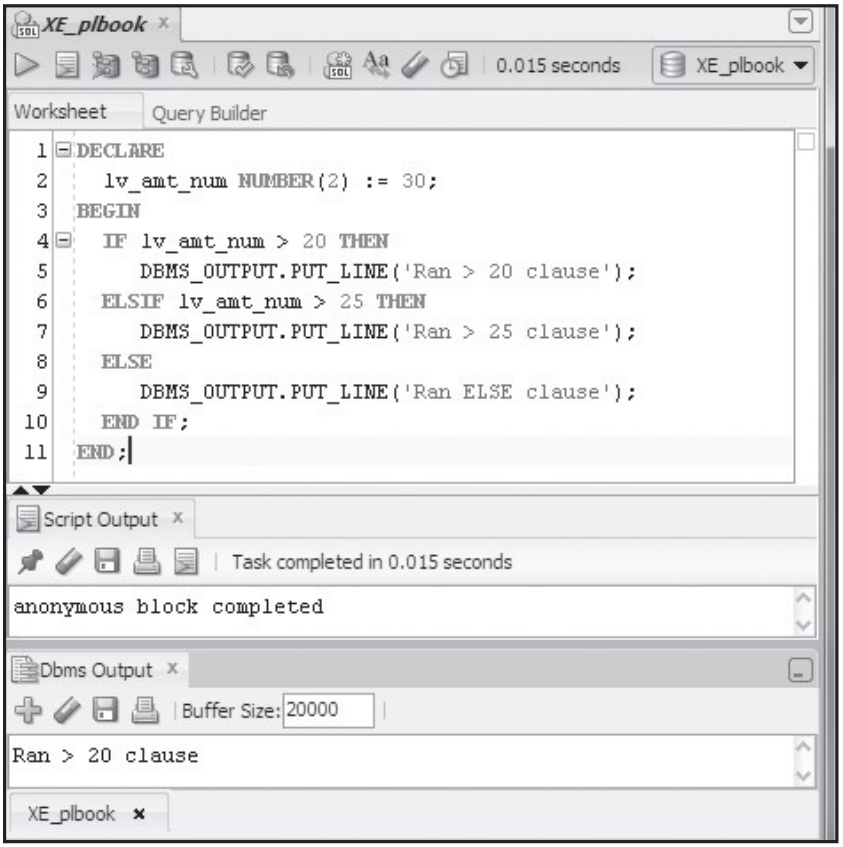


FIGURE 2-21 Testing the order of `IF` conditions

CHALLENGE 2-6

Create a flowchart and block for the following requirements: The block should calculate the discount amount based on a promotion code value. Discounts assigned by a promotion code are A=5%, B=10%, and C=15%. A discount of 2% should be applied if no promotion code is used. Create a variable to hold the promotion code value. Use an order total variable containing a value of $100 to test the block and a variable to hold the discount amount calculated. Run the code block for each possible promotion code value and display the discount amount to test your conditional processing. Next, modify the block so that no discount is applied if a promotion code isn't used. (In other words, the discount should be 0.)

Operators in an `IF` Clause

Keep in mind that more than one condition can be checked in each `IF` clause. The logical operators of `OR` and `AND` used in SQL are also available in PL/SQL. For example, the `IF` clause using an `OR` operator in the following code uses a tax rate of 6% if the shipping state is VA or PA:

```
IF lv_state_txt = 'VA' OR lv_state_txt = 'PA' THEN
  lv_tax_num := lv_sub_num * .06;
ELSE
  lv_tax_num := lv_sub_num * .04;
END IF;
```

Be careful to use complete conditional expressions when using logical operators. A common mistake is using incomplete conditions in the IF clause, as in the first line of the following code. (Notice that the first line of the IF statement uses an incomplete condition with the OR.)

```
IF lv_state_txt = 'VA' OR 'PA' THEN
  lv_tax_num := lv_sub_num * .06;
ELSE
  lv_tax_num := lv_sub_num * .04;
END IF;
```

As part of being familiar with your data, you need to consider whether the values being checked in the IF clauses could be NULL. If an action must occur when the value being checked contains a NULL value, the IF statement must contain an explicit check for a NULL value (IS NULL) or an ELSE clause. In the following code, the ELSE clause runs if the state value is NULL or anything but VA or PA:

```
IF lv_state_txt = 'VA' OR lv_state_txt = 'PA' THEN
  lv_tax_num := lv_sub_num * .06;
ELSE
  lv_tax_num := lv_sub_num * .04;
END IF;
```

However, if you eliminate the ELSE clause but still want a particular action to occur if the state is a NULL value, you must add an ELSIF to check for a NULL value, as shown in the following code:

```
IF lv_state_txt = 'VA' OR lv_state_txt = 'PA' THEN
  lv_tax_num := lv_sub_num * .06;
ELSIF lv_state_txt IS NULL THEN
  lv_tax_num := lv_sub_num * .04;
END IF;
```

This IF statement calculates a 6% tax amount for states of VA or PA and 4% if the state is a NULL value. No activity occurs if the state is any other value.

The IN operator is another method to check several conditions in one IF clause by providing a list of values. The following example uses the IN operator to check for three state values in the first IF clause:

```
IF lv_state_txt IN ('VA','PA','ME') THEN
  lv_tax_num := lv_sub_num * .06;
ELSE
  lv_tax_num := lv_sub_num * .04;
END IF;
```

CHALLENGE 2-7

Create a block to calculate the discount amount based on a promotion code value. A variety of promotion codes are used to identify the source of the promotion, so some of the codes might apply the same percentage discount. Discounts assigned by a promotion code are A=5%, B=10%, C=10%, D=15%, and E=5%. If no promotion code value is indicated, the discount should be 0. Create a variable to hold the promotion code. Use an order total variable containing a value of $100 to test the block and a variable to hold the discount amount calculated. Run the code block for each possible promotion code value, and display the discount amount calculated to test your conditional processing.

Nested IF Statements

As conditional checking becomes more complex in a program, you might need to nest IF statements, which means embedding a complete IF statement inside another IF statement. For example, Brewbean's might want to apply product price discounts based on both the type of product (Equipment or Coffee) and the price of an item. The following example shows the outer IF statement in bold, which checks for the type of product. The inner or nested IF statement runs only when the product type is E (equipment) and checks the price to determine the discount percent to assign. For the product type C (coffee), a discount of .05 is assigned regardless of the product price.

```
IF lv_type_txt = 'E' THEN
  IF lv_price_num > 85 THEN     <-- Inner or nested IF begins
    lv_disc_num = .20;
  ELSIF lv_price_num > 45 THEN
    lv_disc_num = .15;
  ELSE
    lv_disc_num = .10;
  END IF;                       <-- Inner or nested IF ends
ELSIF lv_type_txt = 'C' THEN
  lv_disc_num = .05;
END IF;
```

An alternative approach to the nested IF example is using compound conditions with logical operators, as in the following example. However, this method can lead to inefficient execution. Using this approach, the type variable might need to be checked a number of times before hitting the price level condition that matches. In addition, most programmers find it easier to read and interpret nested statements.

```
IF lv_type_txt = 'E' AND lv_price_num > 85 THEN
  lv_disc_num = .20;
```

CASE Statements

CASE statements are another method for checking conditions. Oracle 9*i* introduced CASE statements to process conditional logic in a manner similar to how IF statements work. CASE statements are available in most programming languages, and many developers are familiar with this type of code. Most programmers choose one method to code decision processing: IF/THEN or CASE statements.

TIP

The addition of the CASE statement doesn't really add functionality in PL/SQL programming; however, advocates of the CASE statement claim it leads to more compact, understandable coding. The choice between IF statements and CASE statements is more a preference of the programming shop, which should establish standard practices to maintain consistency of coding.

The CASE statement begins with the keyword CASE followed by a selector that indicates the value to be checked; typically, it's a variable name. This selector is followed by WHEN clauses to determine which statements should run based on the selector's value.

An ELSE clause can be included at the end, as in the IF statements. The following example shows the syntax for this statement:

```
CASE variable
  WHEN condition THEN statement;
  WHEN condition THEN statement;
  ELSE statement;
END CASE;
```

Review the tax calculation in Figure 2-18. The code in Figure 2-22 performs the same task, except it uses a CASE statement. The CASE statement evaluates the same way the IF statement does, in that it works from the top down until finding a condition that's TRUE. However, the evaluation of the state variable is included only once at the top of the CASE statement instead of with every condition clause in the IF statement.

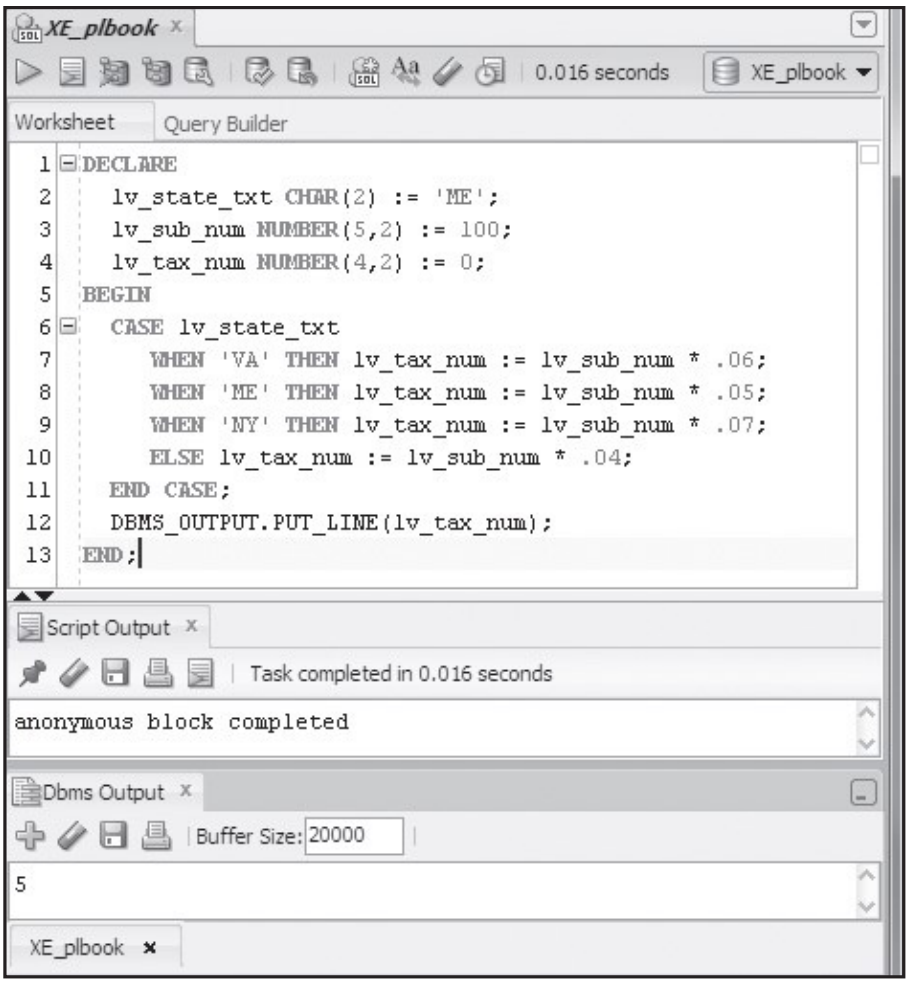


FIGURE 2-22 A basic CASE statement

Refer to Figure 2-18 again. If the ELSE clause is omitted in an IF statement, the IF statement can run and potentially not execute any code. If no matches are found in the IF/ELSIF clauses, the IF statement ends successfully without running any code. This isn't what happens, however, when the ELSE clause is omitted from the CASE statement in Figure 2-22. If no TRUE conditions are found in the WHEN clauses, and an ELSE clause isn't included, the CASE statement includes an implicit ELSE clause that raises an Oracle error. Follow these steps to generate a "CASE not found" error:

1. Enter the PL/SQL block shown in Figure 2-22 and run it.
2. Remove the ELSE clause from the block.
3. Initialize the state variable to the value TX. Notice that the WHEN clauses in the CASE statement don't execute for this state value.
4. Run the modified block. You should get a "CASE not found" error, as shown in Figure 2-23, because no statement is executed by the CASE statement. None of the WHEN clauses address the state value of TX, and the ELSE clause has been removed.

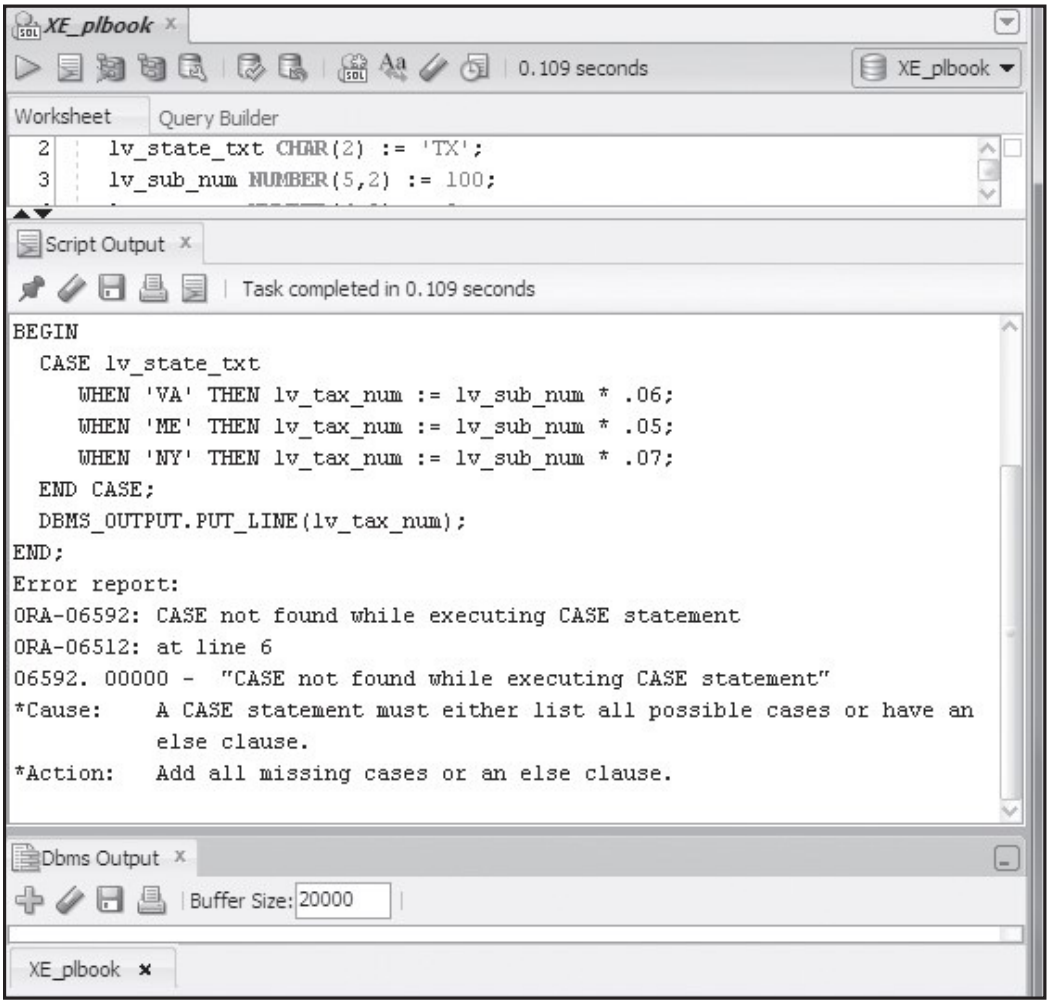


FIGURE 2-23 Error caused when a basic CASE statement doesn't find a WHEN clause match

CHALLENGE 2-8

Create a block that uses a CASE statement to determine the discount amount based on a promotion code value. Discounts assigned by a promotion code are A=5%, B=10%, and C=15%. A discount of 2% should be applied if no promotion code is used. Create a variable to hold the promotion code value. Use an order total variable containing a value of $100 to test the block and a variable to hold the discount amount calculated. Run the code block for each possible promotion code value, and display the discount amount to test your conditional processing.

Searched CASE Statements

Another form of the CASE statement, called a **searched CASE statement**, is available. It doesn't use a selector but evaluates conditions placed in WHEN clauses separately. The structure of a searched CASE statement is quite similar to an IF statement. The conditions checked in WHEN clauses must evaluate to a Boolean value of TRUE or FALSE to allow different items, such as shipping state and zip code, to be checked in the same CASE statement.

To see how a searched CASE statement is used, assume that Brewbean's tax calculation must consider not only the state, but also special rates applied by some localities that can be identified via zip codes. Figure 2-24 shows using a CASE statement to apply a 6% tax rate to all VA residents except those in zip code 23321, which uses a rate of 2%.

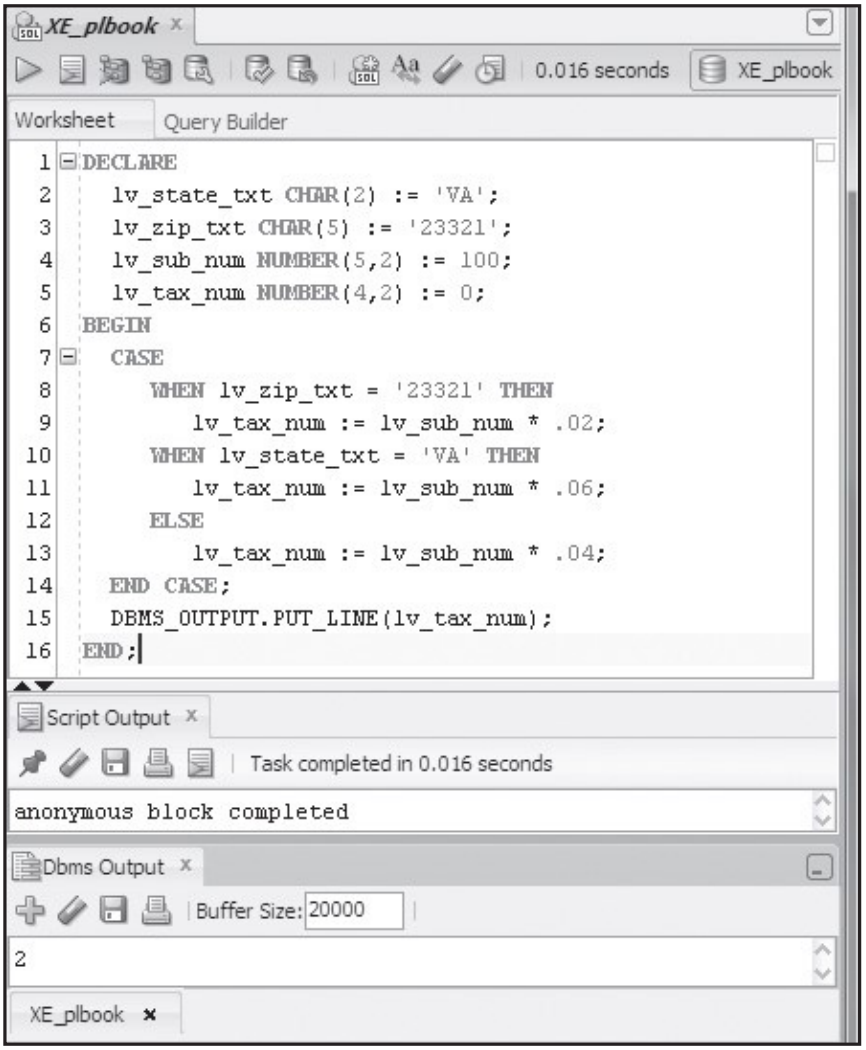


FIGURE 2-24 A searched CASE statement checking different variable values

This block processes with a state of VA and a zip code of 23321. Keep in mind that conditional statements are evaluated from the top down and end when a TRUE condition is discovered. Therefore, processing finds a TRUE condition with the first WHEN clause and uses a tax rate of 2%.

This structure allows checking multiple conditions in each WHEN clause, such as the following example checking price ranges. The conditions checked in the WHEN clause can involve different variables.

```
CASE
  WHEN lv_price_num >= 10 AND lv_price_num < 20 THEN
    lv_disc_num := .05;
  WHEN lv_price_num >= 20 AND lv_price_num < 40 THEN
    lv_disc_num := .10;
  ELSE
    lv_disc_num := 0;
END CASE;
```

Using a CASE Expression

The CASE keyword can also be used as an expression rather than a statement. A **CASE expression** evaluates conditions and returns a value in an assignment statement. For example, in the tax calculation, the end result has been putting a value for the calculated tax amount in a variable. You can also use a CASE expression for this task, as shown in Figure 2-25.

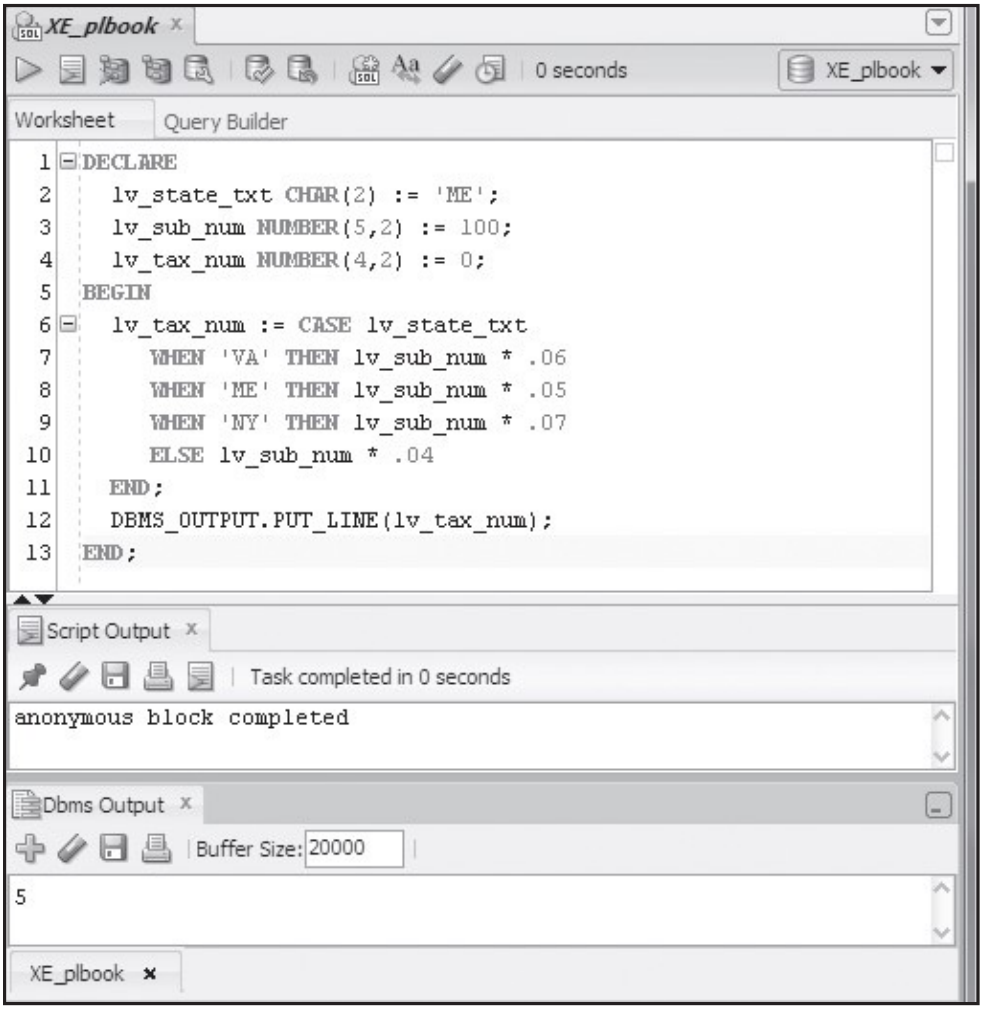


FIGURE 2-25 Using a CASE expression to determine the applicable tax rate

The CASE clause in this figure actually serves as the value expression of an assignment statement. If the goal of your conditional checking is to assign a value to a variable, this format makes this intention clear because it begins with the assignment statement. A CASE expression, even though constructed much the same as a basic CASE statement, has some subtle syntax differences. The WHEN clauses don't end with semicolons, and the

statement doesn't end with END CASE;. (Instead, it ends with END;.) Also, unlike a CASE statement, if a CASE expression has no ELSE clause and no WHEN conditions are matched, the result is a NULL value rather than a "CASE not found" error.

A CASE expression can be structured with more flexibility, as shown in the following code, that allows checking multiple conditions in the WHEN clauses with logical operators. Also, notice that the WHEN clauses assign a specific discount amount, which is then used in the calculation included in the END statement (* lv_sub_num).

```
DECLARE
  lv_state_txt CHAR(2) := 'ME';
  lv_sub_num NUMBER(5,2) := 100;
  lv_tax_num NUMBER(4,2) := 0;
BEGIN
  lv_tax_num :=
    CASE
      WHEN lv_state_txt = 'VA' OR lv_state_txt = 'TX' THEN .06
      WHEN lv_state_txt = 'ME' OR lv_state_txt = 'WY' THEN .05
      ELSE 0
  END * lv_sub_num;
  DBMS_OUTPUT.PUT_LINE(lv_tax_num);
END;
```

Nested CASE Statements

Just as IF statements can be nested, CASE statements can be embedded in each other. As an example, return to the earlier nested IF statement that checked for product type and then checked product price for equipment products. In the following code, the outer CASE statement checks the product type, and the inner CASE statement is executed only if the product type is E. The inner CASE statement checks the price value to set the discount amount. If the product type is C, the inner CASE statement doesn't execute.

```
CASE
  WHEN lv_type_txt = 'E' THEN
    CASE
      WHEN lv_price_num >= 85 THEN
        lv_disc_num := .20;
      WHEN lv_price_num >= 45 THEN
        lv_disc_num := .15;
      ELSE
        lv_disc_num := .10;
    END CASE;
  WHEN lv_type_txt = 'C' THEN
    lv_disc_num := 0;
END CASE;
```

CHALLENGE 2-9

Create a block using a CASE expression to determine the discount amount based on a promotion code value. Discounts assigned by a promotion code are A=5%, B=10%, and C=15%. A discount of 2% should be applied if no promotion code is used. Create a variable to hold the promotion code value. Use an order total variable containing a value of $100 to test the block and a variable to hold the discount amount calculated. Run the code block for each possible promotion code value, and display the discount amount to test your conditional processing.

Conditional IF/THEN and CASE statements are invaluable in creating programming logic; however, additional constructs are needed. The next section introduces looping structures that allow running portions of code more than once.

LOOPING CONSTRUCTS

What if you need to calculate the total tax amount for an order and apply different tax rates on coffee (as a food item) and equipment items? You would need to look at each detailed item (products purchased in an order) to determine the correct tax amount and apply the same logic to each item. If an order contains five items, the same logic that determines the tax amount for each item needs to be processed five times—once for each item. To handle this type of situation more efficiently, you can use **looping constructs**, which make it possible to repeat processing a portion of code. The flowchart in Figure 2-26 illustrates the processing described in this example.

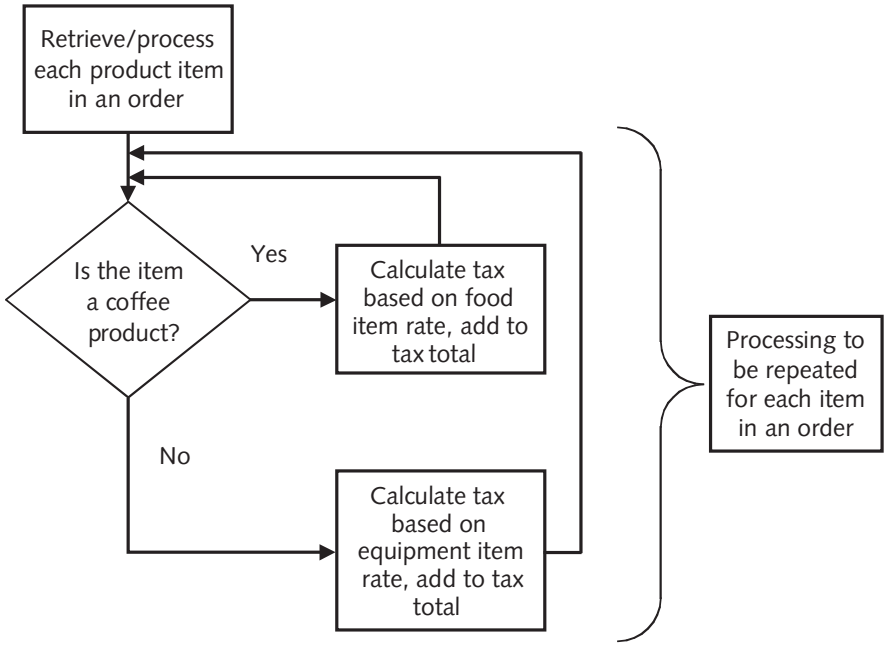


FIGURE 2-26 Flowchart including the tax calculation looping structure

Loops are used when you need to repeat a line or lines of code in a block. In every loop, the system must be instructed on which statements should be repeated and when to end the repeating action or stop the loop. Three types of PL/SQL loops are covered in the following sections: basic, WHILE, and FOR.

Basic Loops

A **basic loop** uses the LOOP and END LOOP markers to begin and end the loop code, which includes any statements to be repeated. One of the most important parts of a loop is making sure something stops the loop iteration. The EXIT WHEN clause must include a condition that evaluates to TRUE at some point, which serves as the instruction to stop the loop. The following code shows the syntax of a basic loop:

```
LOOP
   -- Statements --
   EXIT WHEN condition;
END LOOP;
```

The code in Figure 2-27 contains a basic loop that loops five times. The only statement that runs is the `DBMS_OUTPUT` action, which displays the counter's value onscreen. The counter is established as a mechanism to instruct the loop when to stop repeating. The `lv_cnt_num` variable holds the value 1 on the loop's first iteration. The numbers 1 to 5 are displayed onscreen, and then the loop stops because the `lv_cnt_num` variable holds a value of 5, and the `EXIT WHEN` condition evaluates to `TRUE`.

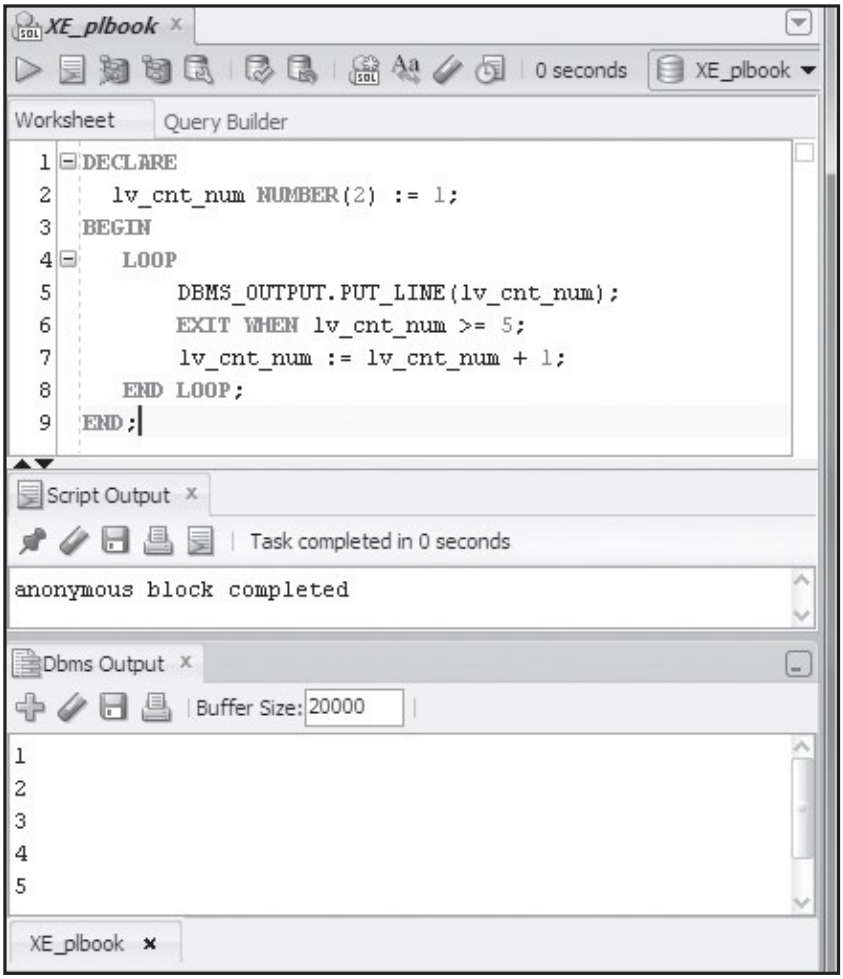


FIGURE 2-27 Using a basic loop

TIP

If the `EXIT WHEN` clause isn't included in the code in Figure 2-27, the result is the programmer's nightmare of an **infinite loop**, a loop that's never instructed to stop and continues looping indefinitely. When this happens, the code can't continue with any processing beyond the loop.

NOTE

Because you haven't processed rows of data from a database yet, the looping examples use scalar variables to show the flow of processing. Retrieving rows of data from a database is covered in subsequent chapters, and looping constructs are used to process multiple rows returned from a database.

To see the effect of the order of loop statements on processing, follow these steps:

1. Enter the PL/SQL block shown in Figure 2-27 with one modification. Move the `EXIT WHEN` line of code to the top of the loop so that it's the first statement in the loop to execute.
2. Run the block and review the output. How did the results change? Only values 1 through 4 are displayed because the loop ends before displaying the value 5.

CHALLENGE 2-10

Create a block that initializes a variable to the value 11. Include a loop to display the variable value and subtract 2 from the variable. End the loop processing when the variable value is less than 2.

Other languages offer variations on loop statements, such as a `LOOP UNTIL`, which evaluates a condition at the bottom of the loop. This bottom-of-the-loop evaluation guarantees that the loop always iterates at least once. PL/SQL doesn't include this type of loop statement, but you can use a basic loop with an `EXIT WHEN` clause at the end of the loop to make sure the loop runs at least one time, as shown in the following code:

```
DECLARE
  lv_cnt_num NUMBER(2) :=1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE(lv_cnt_num);
    lv_cnt_num := lv_cnt_num + 1;
    EXIT WHEN lv_cnt_num >= 5;
  END LOOP;
END;
```

An `EXIT` statement rather than an `EXIT WHEN` condition statement can also be used to end a loop. In the following loop, an `IF` statement is used to check the state of the `lv_cnt_num` value and executes an `EXIT` (ends the loop) if the value is greater than or equal to 5:

```
LOOP
  DBMS_OUTPUT.PUT_LINE(lv_cnt_num);
  IF lv_cnt_num >= 5 THEN
    EXIT;
  ELSE
    lv_cnt_num := lv_cnt_num + 1;
END LOOP;
```

TIP

Multiple `EXIT` statements can be included in a loop as needed. Keep in mind that after any `EXIT` instruction is carried out, processing leaves the loop and continues with the next statement in the block.

WHILE Loops

A **WHILE loop** differs from other types of loops, in that it includes a condition to check at the top of the loop in the LOOP clause. For each iteration of the loop, this condition is checked, and if it's TRUE, the loop continues. If the condition is FALSE, the looping action stops. Ensure that whatever is checked in the loop condition actually changes value as the loop iterates so that at some point, the condition resolves to FALSE and, therefore, ends the looping action. The following example shows the WHILE loop syntax:

```
WHILE condition LOOP
   -- Statements --
END LOOP;
```

Figure 2-28 shows a rewrite of the basic loop example shown in Figure 2-27, except that it uses the WHILE loop format. Notice that this loop iterates only five times, at which time the lv_cnt_num variable holds the value 6. At this point, the WHILE clause runs and determines that the condition evaluates to FALSE (in other words, that 6 isn't <= 5); therefore, the looping action stops. Keep in mind that the condition is evaluated at the top of the loop, which means there's no guarantee the code inside the loop will run at all. If lv_cnt_num has a value of 11 before reaching the loop statement, for example, the WHILE condition evaluates to FALSE, and the looping action never runs.

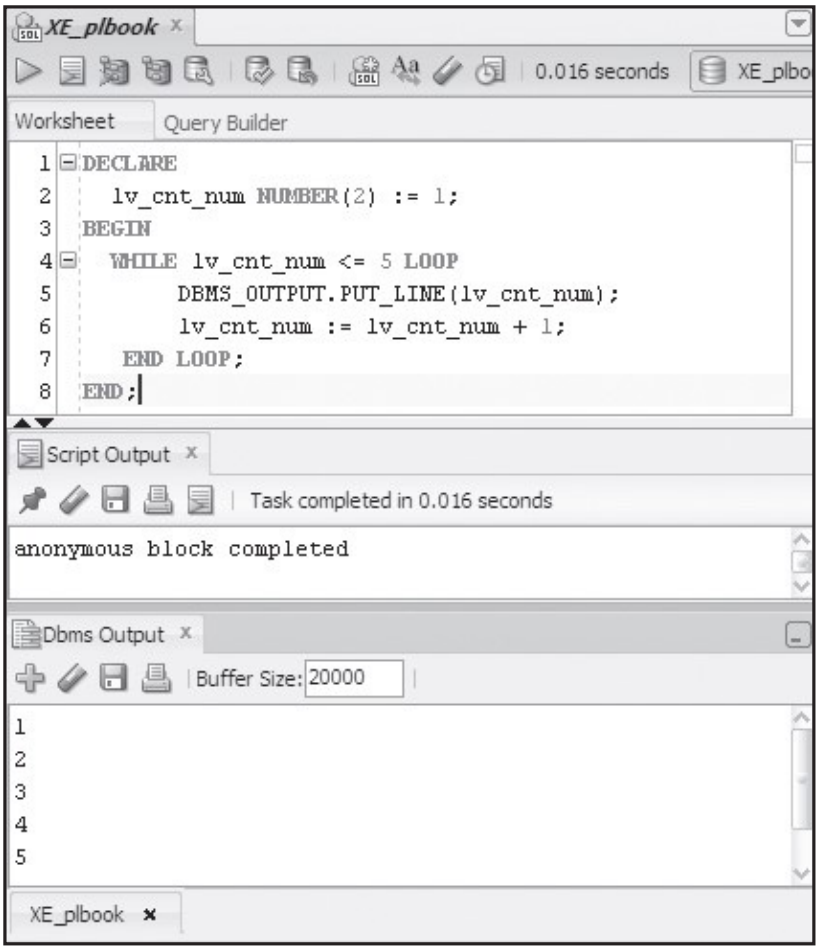


FIGURE 2-28 Using WHILE loop processing

CHALLENGE 2-11

Create the same block described in Challenge 2-10, but use a `WHILE` loop instead of a basic loop.

`FOR` Loops

A **`FOR` loop** performs the same job of iterating; however, this type of loop indicates how many times to loop by including a range in the statement. The following example shows the syntax of the `FOR` loop:

```
FOR counter IN lower_bound..upper_bound LOOP
  -- Statements --
END LOOP;
```

The counter is a variable that holds the value of the current iteration number and is automatically incremented by 1 each time the loop iterates. It begins with the value supplied for the *`lower_bound`*, and the iterations continue until the counter reaches the value supplied for the *`upper_bound`*. Even though any name can be used for the counter, using `i` is typical. The lower and upper bounds can be numbers or variables containing numeric values.

By indicating a numeric range, the `FOR` loop specifies how many times the loop in the opening `LOOP` clause should run. Figure 2-29 shows a `FOR` loop, which produces the same results as the counter examples used with the basic and `WHILE` loops. The range is indicated in the `FOR` clause on Line 2.

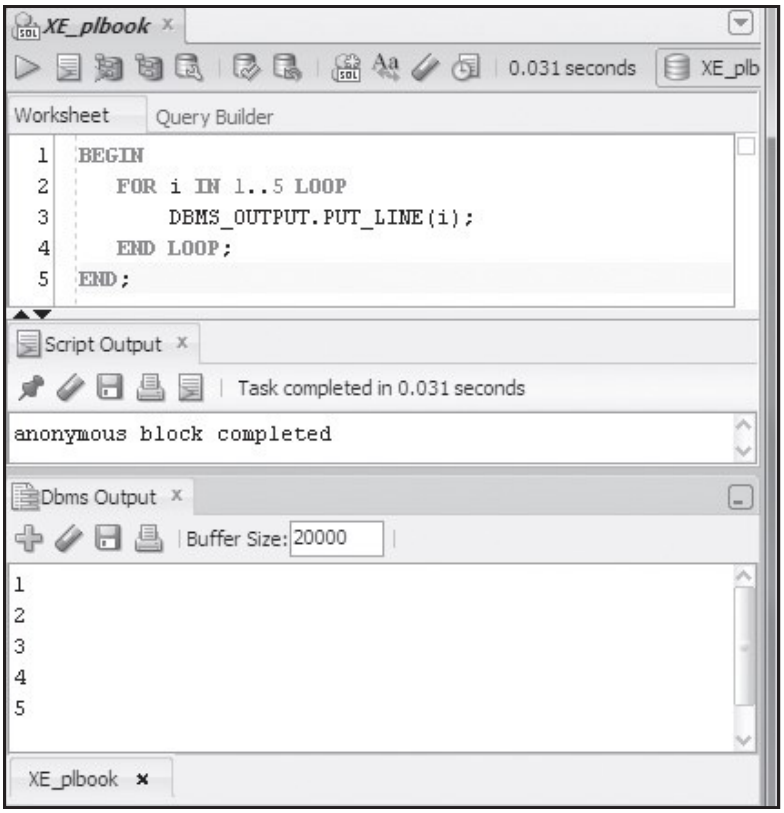


FIGURE 2-29 Using a `FOR` loop

Several tasks are performed in the `FOR` clause that starts the loop in Figure 2-29. First, the `FOR` clause sets up a counter variable automatically. In this example, the counter is named `i`. The second task is setting up a range of values for the counter variable that

control the number of times the loop runs (`1..5`). In this example, the counter (`i`) holds the value 1 in the loop's first iteration. Each iteration increments the counter by 1 automatically, and the loop stops after running five repetitions.

The loop range must indicate a lower bound and upper bound value that determines the number of times the loop iterates. The values can be provided in the form of numbers, variables, or expressions evaluating to numeric values. By default, the counter starts at the lower bound value and is incremented by 1 for each loop iteration. The loop in this example always iterates five times because the range is indicated with numeric values. However, a variable could be used in the range to make the number of iterations more dynamic. That is, each time the block containing the loop is run, the loop might iterate a different number of times. For example, the number of iterations of the `FOR` loop in the following block is determined by the value of `lv_upper_num`:

```
DECLARE
  lv_upper_num NUMBER(3) := 3;
BEGIN
  FOR i IN 1..lv_upper_num LOOP
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
```

Instead of being initialized to a value, the `lv_upper_num` variable could be populated with a value at runtime (which you do in Chapters 3 and 4), which could be different for each execution. The counter's value can be referenced inside the loop and become an integral value in your processing. In this example, the `i` value is displayed onscreen. The counter variable (`i`) can be referenced in the loop but can't be assigned a value because the loop controls the counter's value.

The `REVERSE` option is also available to force the counter to begin with the range's upper bound and increment by -1 in each loop iteration until the lower bound is reached. The `REVERSE` keyword should be included in the `FOR LOOP` clause immediately before the range, as shown in the following code:

```
FOR i IN REVERSE 1..5 LOOP
```

In many languages, programmers can indicate the increment value in the loop. For example, an increment value of 2 causes the counter variable to increase by 2 instead of 1 in each loop iteration. This feature isn't available as an option in the `FOR LOOP` statement, which forces the increment value to be 1. Other languages have a `STEP` option for specifying an increment value. However, you can manipulate the counter value to simulate an increment value greater than 1 by adding code to the loop logic. Say you want a loop that simulates an increment value of 5. In the following block, you add a multiplier in the loop statements. The loop runs only twice; however, the values displayed are 5 and 10 because a multiplier of 5 is used with the counter variable in Line 3:

```
BEGIN
  FOR i IN 1..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i*5);
  END LOOP;
END;
```

CHALLENGE 2-12

Create a block that initializes a variable to the value 11. Use a `FOR` loop to display the variable value and subtract 2 from the variable. Set the loop to iterate five times.

CONTINUE Statements

The CONTINUE statement introduced in Oracle 11g provides a mechanism for exiting a loop's current iteration. It doesn't end loop processing, as the EXIT statement does; it simply moves loop processing to the next iteration. This statement has two forms: CONTINUE and CONTINUE WHEN. Try using this feature to create a loop that executes only every fifth iteration. The output in Figure 2-30 confirms that the CONTINUE WHEN statement at the beginning of the loop instructs the loop to move to the next iteration if the counter value can't be evenly divided by 5 (determined with the MOD function). If the loop counter can be evenly divided by 5, the remaining statements in the loop run.

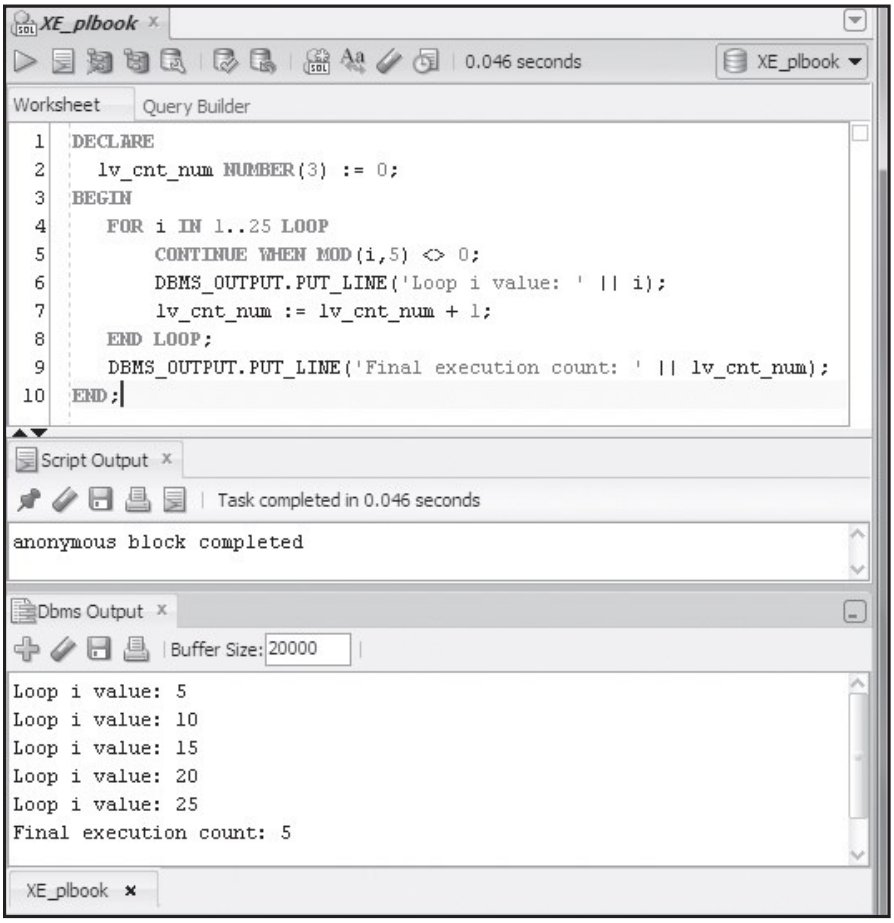


FIGURE 2-30 Using a CONTINUE WHEN statement

Common Errors in Using Looping Statements

Now that you have looked at all three loop types, it's time to discuss a couple of factors to consider when creating loops.

EXIT Clause in Loops

One caution concerns using the EXIT clause in loops. Even though it can be used in any type of loop to stop the looping action, it's considered good form to use it only in basic loops. The WHILE and FOR loops are constructed with conditions in the LOOP statement to determine when the looping action begins and ends. Using an EXIT clause can circumvent these conditions and stop the looping at a different point in the processing. This type of loop is both hard to read and debug.

Static Statements

A second caution is to remember that loops execute all the statements they contain in each iteration. To keep code efficient and minimize statement processing, any statements that are static in nature should be placed outside a loop. For example, say you have a loop containing a calculation that tallies up values, as shown in the following code:

```
DECLARE
  lv_upper_num NUMBER(3) := 3;
  lv_total_num NUMBER(3);
BEGIN
  FOR i IN 1..lv_upper_num LOOP
    lv_total_num := lv_total_num + i;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(lv_total_num);
END;
```

The loop calculates the total, which is stored in the `lv_total_num` variable. If only the final total needs to be displayed, the `DBMS_OUTPUT` statement is moved outside the loop so that it processes only once, after the loop, as shown in the preceding code.

Nested Loops

Similar to `IF` and `CASE` statements, loops can be nested, too. Figure 2-31 shows a `FOR` loop nested inside another `FOR` loop. The output confirms that for each iteration of the outer loop, the inner loop iterates twice.

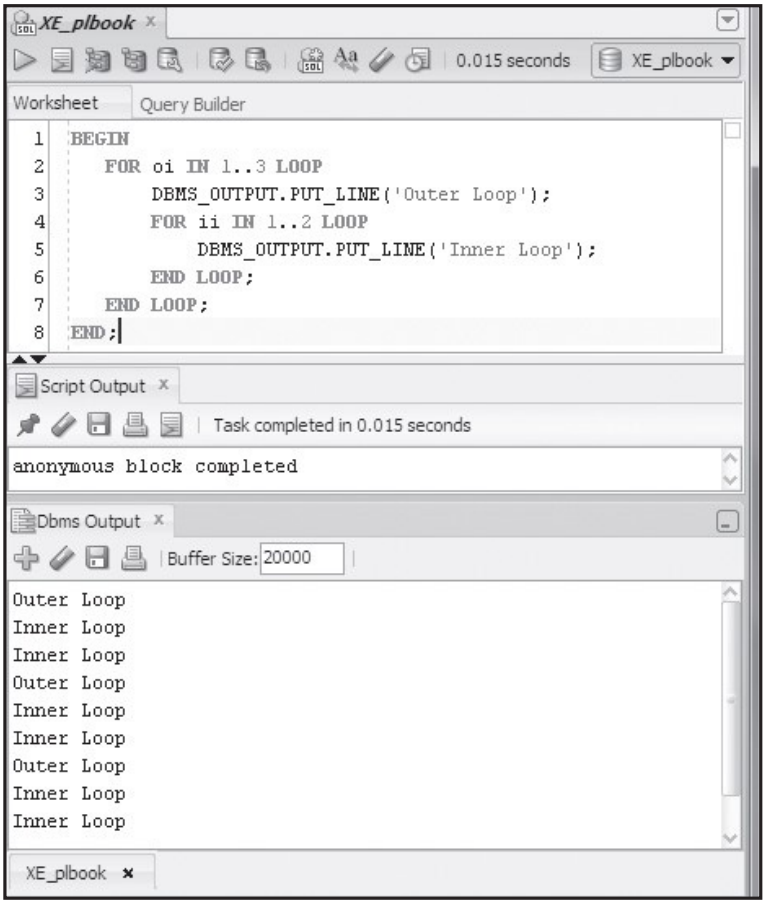


FIGURE 2-31 Nesting `FOR` loops

Nesting loops can be controlled by adding labels to each loop. The following block assigns labels to each loop (such as `<<outer>>`). The `IF` statement in the inner loop is checking the counter value for the outer loop (`oi`) and instructs the inner loop to execute only if it's 2. Otherwise, return to the outer loop as instructed with the `CONTINUE` statement referring to the outer loop's label. Labels can be referenced to control processing flow, and they also make code more readable.

```
BEGIN
  <<outer>>
  FOR oi IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE('Outer Loop');
    <<inner>>
    FOR ii IN 1..2 LOOP
      IF oi <> 2 THEN CONTINUE outer; END IF;
      DBMS_OUTPUT.PUT_LINE('Inner Loop');
    END LOOP;
  END LOOP;
END;
```

The output for this block is as follows:

```
Outer Loop
Outer Loop
Inner Loop
Inner Loop
Outer Loop
```

Chapter Summary

- Flowcharts help developers map out the sequence of events to be coded.
- A decision structure identifies different actions that occur during execution, depending on values at runtime.
- Looping structures allow repeating code execution.
- A PL/SQL block can contain DECLARE, BEGIN, EXCEPTION, and END sections. The BEGIN and END sections are required.
- Variables are named memory areas that hold values to allow retrieving and manipulating values in your programs.
- Variables to store and manipulate values in a block are created in the DECLARE section.
- Scalar variables can hold a single value. Common data types are `VARCHAR2`, `CHAR`, `NUMBER`, `DATE`, and `BOOLEAN`.
- Variable-naming rules are the same as those for database objects, including beginning with an alpha character, and can contain up to 30 characters.
- At a minimum, variable declarations must include a name and data type.
- Variables can be initialized with a value in the DECLARE section, using the PL/SQL assignment operator.
- A `NOT NULL` option can be used in a variable declaration to require that the variable always contains a value. These variables must be initialized.
- A `CONSTANT` option can be used in a variable declaration to enforce that the value can't be changed in the block. These variables must be initialized.
- The `DBMS_OUTPUT.PUT_LINE` statement is used to display values onscreen.
- Calculations can be performed with scalar variables.
- SQL single-row functions can be used in PL/SQL statements.
- `IF` statements are a decision-making structure to control program execution based on runtime values. They use the structure of `IF/THEN/ELSIF/ELSE`.
- `CASE` statements are another method for performing decision making in PL/SQL. A searched `CASE` statement allows evaluating different values in an operation.
- A basic loop structure needs an `EXIT WHEN` statement to instruct the system when to stop repeating the code.
- A `WHILE` loop includes a condition in its opening statement that determines whether the loop executes.
- A `FOR` loop includes a range in its opening statement to control how many times the loop repeats.
- A `CONTINUE` statement can be used to move to the next iteration of a loop.
- Use caution when incorporating an `EXIT` clause and placing static statements in looping constructs because doing so can lead to unexpected or inefficient results.
- `IF`, `CASE`, and looping constructs can be nested.

Review Questions

1. Which of the following variable declarations is illegal?
   a. `lv_junk NUMBER(3);`
   b. `lv_junk NUMBER(3) NOT NULL;`
   c. `lv_junk NUMBER(3) := 11;`
   d. `lv_junk NUMBER(3) CONSTANT := 11;`
2. Which of the following is *not* a possible value for a Boolean variable?
   a. `TRUE`
   b. `FALSE`
   c. `BLANK`
   d. `NULL`
3. What type of variable can store only one value?
   a. local
   b. scalar
   c. simple
   d. declared
4. What keyword is used to check multiple conditions with an `IF` statement?
   a. `ELSE IF`
   b. `ELSEIF`
   c. `ELSIF`
   d. `ELSIFS`
5. What type of loop can be used if the loop might not need to execute under certain circumstances?
   a. `FOR`
   b. `WHILE`
   c. basic
   d. All of the above
6. How is the looping action of a basic loop stopped?
   a. It's stopped when the condition in the `LOOP` statement is `FALSE`.
   b. This type of loop has a predetermined number of loops to complete.
   c. The condition in an `EXIT WHEN` statement is `FALSE`.
   d. The condition in an `EXIT WHEN` statement is `TRUE`.
7. When does a `WHILE` loop evaluate the condition that determines whether the looping action continues?
   a. at the beginning of the loop
   b. somewhere inside the loop
   c. at the end of the loop
   d. all of the above

8. If you know the number of loop iterations ahead of time, what type of loop should be used?
   a. `FOR`
   b. `WHILE`
   c. basic
   d. none of the above
9. What commands can be used to end loop execution? (Choose all that apply.)
   a. `CONTINUE`
   b. `EXIT`
   c. `EXIT WHEN`
   d. `STOP`
10. Which programming constructs can use a `CONTINUE` statement?
    a. `IF/THEN` statements
    b. loops
    c. `CASE` statements
    d. all of the above
11. What are variables, and why are they needed?
12. Name the three main types of loop structures in PL/SQL, and explain the difference in how each determines how many times a loop iterates.
13. What are the two types of decision structures in PL/SQL?
14. How can flowcharts assist developers?
15. What happens when a `CONSTANT` option is set in a variable declaration?

Advanced Review Questions

1. Review the following block. What value is displayed by the `DBMS_OUTPUT` statement?

```
DECLARE
  lv_junk1 CHAR(1) := 'N';
  lv_junk2 CHAR(1) := 'N';
BEGIN
  lv_junk1 := 'Y';
  DBMS_OUTPUT.PUT_LINE(lv_junk2);
END;
```

   a. Y
   b. N
   c. NULL
   d. This block raises an error.
2. Review the following `IF` statement. What is the resulting value of `lv_ship_num` if `lv_amt_num` has the value 1200?

```
IF lv_amt_num > 500 THEN
  lv_ship_num := 5;
```