CHAPTER 5

Using SQL*Plus

In this chapter, you will

- View the structure of a table
- Edit a SQL statement
- Save and run scripts containing SQL statements and SQL*Plus commands
- Format column output
- Define and use variables
- Create simple reports

In the last section of this chapter, you'll also learn how to write SQL statements that generate other SQL statements. Let's plunge in and examine how you view the structure of a table.

Viewing the Structure of a Table

You use the `DESCRIBE` command to view the structure of a table. You can save some typing by shortening the `DESCRIBE` command to `DESC` (`DESC[RIBE]`). Knowing the structure of a table is useful because you can use the information to formulate a SQL statement. For example, you can figure out the columns you want to query in a `SELECT` statement.

NOTE
*You typically omit the semicolon character (;) when issuing SQL*Plus commands.*

The following example uses the `DESCRIBE` command to view the structure of the `customers` table; notice that the semicolon character (;) is omitted from the end of the command:

```
SQL> DESCRIBE customers
Name                      Null?          Type
 --------------------- -------- --------------
 CUSTOMER_ID           NOT NULL     NUMBER(38)
 FIRST_NAME            NOT NULL   VARCHAR2(10)
 LAST_NAME             NOT NULL   VARCHAR2(10)
 DOB                                      DATE
 PHONE                            VARCHAR2(12)
```

As you can see from this example, the output from the `DESCRIBE` command has three columns that show the structure of the database table:

- **Name** Lists the names of the columns contained in the table. In the example, you can see the `customers` table has five columns: `customer_id`, `first_name`, `last_name`, `dob`, and `phone`.

- **Null?** Indicates whether the column can store null values. If NOT NULL, the column cannot store nulls. If blank, the column can store null values. In the example, you can see the customer_id, first_name, and last_name columns cannot store null values, but the dob and phone columns can store null values.
- **Type** Indicates the type of the column. In the example, you can see the type of the customer_id column is NUMBER(38) and the type of the first_name is VARCHAR2(10).

The next example uses the shortened DESC command to view the structure of the products table:

```
SQL> DESC products
 Name                      Null?           Type
 --------------------- -------- --------------
 PRODUCT_ID               NOT NULL     NUMBER(38)
 PRODUCT_TYPE_ID                       NUMBER(38)
 NAME                     NOT NULL   VARCHAR2(30)
 DESCRIPTION                         VARCHAR2(50)
 PRICE                                NUMBER(5,2)
```

Editing SQL Statements

As you may have noticed, it becomes tedious to have to repeatedly type similar SQL statements into SQL*Plus. You will be pleased to know SQL*Plus stores your previous SQL statement in a buffer. You can then edit the lines that make up your SQL statement stored in the buffer. Some of the editing commands are listed in the following table. Notice the optional part of each command indicated using square brackets; for example, you can abbreviate the APPEND command to A.

Command	Description
A[PPEND] *text*	Appends *text* to the current line.
C[HANGE] /*old*/*new*	Changes the text specified by *old* to *new* in the current line.
CL[EAR] BUFF[ER]	Clears all lines from the buffer.
DEL	Deletes the current line.
DEL *x*	Deletes the line specified by the line number *x* (line numbers start with 1).
L[IST]	Lists all the lines in the buffer.
L[IST] *x*	Lists line number *x*.
R[UN] or /	Runs the statement stored in the buffer. You can also use / to run the statement.
x	Makes the line specified by the line number *x* the current line.

Let's take a look at some examples of using the SQL*Plus editing commands. First, enter the following SELECT statement into SQL*Plus:

```
SQL> SELECT customer_id, first_name, last_name
  2  FROM customers
  3  WHERE customer_id = 1;
```

SQL*Plus automatically increments and displays the line number when your SQL statement spans more than one line. Make line 1 the current line by entering 1 at the prompt:

```
SQL> 1
  1* SELECT customer_id, first_name, last_name
```

Notice that SQL*Plus displays the current line. Add the dob column to the list of columns to retrieve using the APPEND command:

```
SQL> APPEND , dob
  1* SELECT customer_id, first_name, last_name, dob
```

Next, list all the lines in the buffer using the LIST command:

```
SQL> LIST
  1  SELECT customer_id, first_name, last_name, dob
  2  FROM customers
  3* WHERE customer_id = 1
```

Notice that the current line has been changed to the last line, as indicated by the asterisk character (*). Change the final line to select the customer where the customer_id column is 2 using the CHANGE command. Notice that the line that has been changed is displayed after the command is run:

```
SQL> CHANGE /customer_id = 1/customer_id = 2
  3* WHERE customer_id = 2
```

Finally, execute the query using the RUN command. Notice that the text of the query is repeated before the returned row:

```
SQL> RUN
  1  SELECT customer_id, first_name, last_name, dob
  2  FROM customers
  3* WHERE customer_id = 2

CUSTOMER_ID FIRST_NAME LAST_NAME  DOB
----------- ---------- ---------- ---------
          2 Cynthia    Green      05-FEB-68
```

You can also use a forward slash character (/) to run the SQL statement stored in the buffer. For example:

```
SQL> /

CUSTOMER_ID FIRST_NAME LAST_NAME  DOB
----------- ---------- ---------- ---------
          2 Cynthia    Green      05-FEB-68
```

Saving, Retrieving, and Running Files

SQL*Plus allows you save, retrieve, and run text files containing SQL*Plus commands and SQL statements. You've already seen one example of running a SQL*Plus script: you saw how to run the `store_schema.sql` script file in Chapter 1, which created the `store` schema.

Some of the file commands are listed in the following table.

Command	Description
`SAV[E]` *filename* `[{ REPLACE | APPEND }]`	Saves the contents of the SQL*Plus buffer to a file specified by *filename*. You can append the content of the buffer to an existing file using the `APPEND` option. You can also overwrite an existing file using the `REPLACE` option.
`GET` *filename*	Retrieves the contents of the file specified by *filename* into the SQL*Plus buffer.
`STA[RT]` *filename*	Retrieves the contents of the file specified by *filename* into the SQL*Plus buffer, and then attempts to run the contents of the buffer.
`@` *filename*	Same as the `START` command.
`ED[IT]`	Copies the contents of the SQL*Plus buffer to a file named `afiedt.buf` and then starts the default editor for the operating system. When you exit the editor, the contents of your edited file are copied to the SQL*Plus buffer.
`ED[IT]` *filename*	Same as the `EDIT` command, but you can specify a file to start editing. You specify the file to edit using the *filename* parameter.
`SPO[OL]` *filename*	Copies the output from SQL*Plus to the file specified by *filename*.
`SPO[OL] OFF`	Stops the copying of output from SQL*Plus to the file, and closes that file.

Let's take a look at some examples of using these SQL*Plus commands. First, enter the following SQL statement into SQL*Plus:

```
SQL> SELECT customer_id, first_name, last_name
  2  FROM customers
  3  WHERE customer_id = 1;
```

Save the contents of the SQL*Plus buffer to a file named `cust_query.sql` using the `SAVE` command:

```
SQL> SAVE cust_query.sql
Created file cust_query.sql
```

NOTE
By default the `cust_query.sql` file is stored in the `bin` subdirectory where you installed the Oracle software.

Retrieve the contents of the `cust_query.sql` file using the `GET` command:

```
SQL> GET cust_query.sql
  1  SELECT customer_id, first_name, last_name
  2  FROM customers
  3* WHERE customer_id = 1
```

Run the contents of the `cust_query.sql` file using the `START` command:

```
SQL> START cust_query.sql

CUSTOMER_ID FIRST_NAME LAST_NAME
----------- ---------- ----------
          1 John       Brown
```

Edit the contents of the SQL*Plus buffer using the `EDIT` command:

```
SQL> EDIT
```

The `EDIT` command starts the default editor for your operating system. On Windows the default editor is Notepad, and on Unix or Linux the default editor is vi or emacs. You can set the default editor using the `DEFINE` command:

```
DEFINE _EDITOR = 'editor'
```

where *`editor`* is the name of your preferred editor.

For example, the following command sets the default editor to vi:

```
DEFINE _EDITOR = 'vi'
```

Figure 5-1 shows the contents of the SQL*Plus buffer in Notepad. Notice that the SQL statement is terminated using a slash character (`/`) rather than a semicolon.

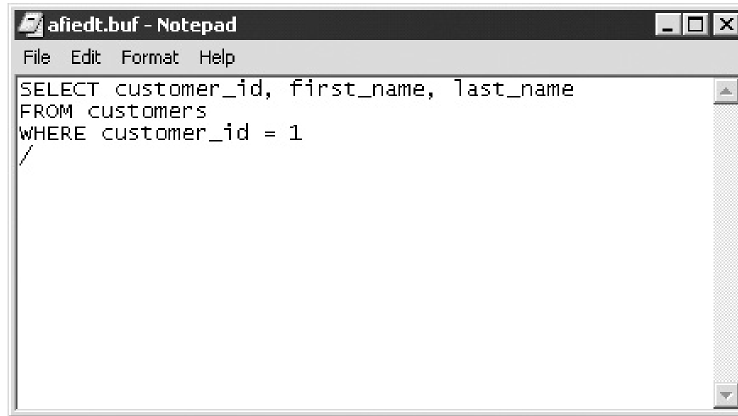

FIGURE 5-1. *Editing the SQL*Plus buffer contents using Notepad*

In your editor, change the `WHERE` clause to `WHERE customer_id = 2` and save and quit from your editor. For example, in Notepad you select File | Exit to quit Notepad; click Yes to save your query when prompted by Notepad. SQL*Plus displays the following output containing your modified query. Notice that the `WHERE` clause has been changed:

```
Wrote file afiedt.buf

  1  SELECT customer_id, first_name, last_name
  2  FROM customers
  3* WHERE customer_id = 2
```

Run your modified query using the slash character (`/`):

```
SQL> /

CUSTOMER_ID FIRST_NAME LAST_NAME
----------- ---------- ----------
          2 Cynthia    Green
```

Next, use the `SPOOL` command to copy the output from SQL*Plus to a file named `cust_results.txt`, run your query again, and then turn spooling off by executing `SPOOL OFF`:

```
SQL> SPOOL cust_results.txt
SQL> /

CUSTOMER_ID FIRST_NAME LAST_NAME
----------- ---------- ----------
          2 Cynthia    Green

SQL> SPOOL OFF
```

Feel free to examine the `cust_results.txt` file; it will contain the previous output between the slash (/) and `SPOOL OFF`. By default, this file is stored in the directory where the Oracle software is installed under the `bin` subdirectory. You can specify the full directory path where you want the file to be written using the `SPOOL` command by adding a directory path to your filename. For example:

```
SPOOL C:\my_files\spools\cust_results.txt
```

Formatting Columns

You use the `COLUMN` command to format the display of column headings and column data. The simplified syntax for the `COLUMN` command is as follows:

```
COL[UMN] {column | alias} [options]
```

where

- *column* specifies the column name.
- *alias* specifies the column alias to be formatted. In Chapter 2 you saw that you can "rename" a column using a column alias; you can then reference your alias in the `COLUMN` command.
- *options* specifies one or more options to be used to format the column or alias.

There are a number of options you can use with the `COLUMN` command. The following table shows some of these options.

Option	Description
FOR[MAT] *format*	Sets the format for the display of the column or alias to that specified in the *format* string.
HEA[DING] *heading*	Sets the text for the heading of the column or alias to that specified in the *heading* string.
JUS[TIFY] [{ left | center | right }]	Aligns the column output to the left, center, or right.
WRA[PPED]	Wraps the end of a string onto the next line of output. This option may cause individual words to be split across multiple lines.
WOR[D_WRAPPED]	Similar to the `WRAPPED` option except that individual words aren't split across two lines.
CLE[AR]	Clears any formatting of columns (sets the formatting back to the default).

The *format* string in the previous table may take a number of formatting parameters. The parameters you specify depend on the data stored in your column:

- If your column contains characters, you can use `Ax` to format the characters, where *x* specifies the width for the characters. For example, `A12` sets the width to 12 characters.
- If your column contains numbers, you can use any of the number formats shown in Table 3-4 of Chapter 3. For example, `$99.99` sets the format to a dollar sign, followed by two digits, the decimal point, and another two digits.
- If your column contains a date, you can use any of the date formats shown in Table 4-2 of Chapter 4. For example, `MM-DD-YYYY` sets the format to a two-digit month (`MM`), a two-digit day (`DD`), and a four-digit year (`YYYY`).

Let's consider an example. You're going to format the output of a query that retrieves the `product_id`, `name`, `description`, and `price` columns from the `products` table. The display requirements, the format strings, and the `COLUMN` commands are shown in the following table.

Column	Display Requirement	Format	`COLUMN` Command
`product_id`	Two digits	99	`COLUMN product_id FORMAT 99`
`name`	Thirteen-character word-wrapped strings and change heading to `PRODUCT_NAME`	A13	`COLUMN name HEADING PRODUCT_NAME FORMAT A13 WORD_WRAPPED`
`description`	Thirteen-character word-wrapped strings	A13	`COLUMN description FORMAT A13 WORD_WRAPPED`
`price`	Dollar symbol, with two digits to the right of the decimal point and two digits to the left of the decimal point	$99.99	`COLUMN price FORMAT $99.99`

Enter the following `COLUMN` commands into SQL*Plus in preparation for executing a query against the `products` table:

```
SQL> COLUMN product_id FORMAT 99
SQL> COLUMN name HEADING PRODUCT_NAME FORMAT A13 WORD_WRAPPED
SQL> COLUMN description FORMAT A13 WORD_WRAPPED
SQL> COLUMN price FORMAT $99.99
```

Next, run the following query to retrieve some rows from the products table. Notice the formatting of the columns in the output due to the previous COLUMN commands:

```
SQL> SELECT product_id, name, description, price
  2  FROM products
  3  WHERE product_id < 6;
```

```
PRODUCT_ID PRODUCT_NAME  DESCRIPTION      PRICE
---------- ------------- ------------- --------
         1 Modern        A description   $19.95
           Science       of modern
                         science

         2 Chemistry     Introduction    $30.00
                         to Chemistry

         3 Supernova     A star          $25.99
                         explodes

         4 Tank War      Action movie    $13.95

PRODUCT_ID PRODUCT_NAME  DESCRIPTION      PRICE
---------- ------------- ------------- --------
                         about a
                         future war

         5 Z Files       Series on       $49.99
                         mysterious
                         activities
```

This output is readable, but wouldn't it be nice if you could just display the headings once at the top? You can do that by setting the page size.

Setting the Page Size

You set the number of lines in a page using the `SET PAGESIZE` command. This command sets the number of lines that SQL*Plus considers one "page" of output, after which SQL*Plus will display the headings again.

Set the page size to 100 lines using the following `SET PAGESIZE` command and run your query again using `/`:

```
SQL> SET PAGESIZE 100
SQL> /

PRODUCT_ID PRODUCT_NAME  DESCRIPTION      PRICE
---------- ------------- ------------- --------
         1 Modern        A description   $19.95
           Science       of modern
                         science

         2 Chemistry     Introduction    $30.00
                         to Chemistry

         3 Supernova     A star          $25.99
                         explodes

         4 Tank War      Action movie    $13.95
                         about a
```

```
                              future war

         5 Z Files            Series on       $49.99
                              mysterious
                              activities
```

Notice the headings are only shown once at the top, and the resulting output looks better.

NOTE
The maximum number for the page size is 50,000.

Setting the Line Size

You set the number of characters in a line using the `SET LINESIZE` command. Set the line size to 50 lines using the following `SET LINESIZE` command and run the new query shown in the following example:

```
SQL> SET LINESIZE 50
SQL> SELECT * FROM customers;

CUSTOMER_ID FIRST_NAME LAST_NAME  DOB
----------- ---------- ---------- ---------
PHONE
------------
          1 John       Brown      01-JAN-65
800-555-1211

          2 Cynthia    Green      05-FEB-68
800-555-1212

          3 Steve      White      16-MAR-71
800-555-1213

          4 Gail       Black
800-555-1214

          5 Doreen     Blue       20-MAY-70
```

The lines don't span more than 50 characters.

NOTE
The maximum number for the line size is 32,767.

Clearing Column Formatting

You clear the formatting for a column using the `CLEAR` option of the `COLUMN` command. For example, the following `COLUMN` command clears the formatting for the `product_id` column:

```
SQL> COLUMN product_id CLEAR
```

You can clear the formatting for all columns using CLEAR COLUMNS. For example:

```
SQL> CLEAR COLUMNS
columns cleared
```

Once you've cleared your columns, the output from queries will use the default format for the columns.

Using Variables

In this section, you'll see how to create variables that may be used in place of actual values in SQL statements. These variables are known as *substitution variables* because they are used as substitutes for values. When you run your SQL statement, you enter values for your variables and those values are then substituted into your SQL statement.

There are two basic types of variables you can use in SQL*Plus:

- **Temporary variables** A temporary variable is only valid for the SQL statement in which it is used and doesn't persist.
- **Defined variables** A defined variable persists until you explicitly remove it, redefine it, or exit SQL*Plus.

You'll learn how to use these types of variables in this section.

Why Are Variables Useful?

Variables are useful because you can create scripts that a user who doesn't know SQL can run. Your script would prompt the user to enter the value for a variable and use that value in a SQL statement. Let's take a look at an example.

Suppose you wanted to create a script for a user who doesn't know SQL, but who wants to see the details of a single specified product in the store. To do this, you could hard code the product_id value in the WHERE clause of a SELECT statement and place that SELECT statement in a SQL*Plus script. For example, the following SELECT statement retrieves product #1:

```
SELECT product_id, name, price
FROM products
WHERE product_id = 1;
```

This query works, but it only retrieves that one product. What if you wanted to change the product_id value to retrieve a different row? You could modify the script, but this would be tedious. Wouldn't it be great if you could supply a variable for the product_id column in the WHERE clause when the query is actually run? A variable would enable you to write a general SQL statement that would work for any product, and the user would simply enter the value for that variable.

Temporary Variables

You define a temporary variable using the ampersand character (&) in a SQL statement, followed by the name you want to call your variable. For example, &product_id_var defines a variable named product_id_var.

When you run the following SELECT statement, SQL*Plus prompts you to enter a value for product_id_var and then uses that variable's value in the WHERE clause of the SELECT statement. If you enter the value 2 for product_id_var, the details for product #2 will be displayed.

```
SQL> SELECT product_id, name, price
  2  FROM products
  3  WHERE product_id = &product_id_var;
Enter value for product_id_var: 2
old   3: WHERE product_id = &product_id_var
new   3: WHERE product_id = 2

PRODUCT_ID NAME                                PRICE
---------- ------------------------------ ----------
         2 Chemistry                              30
```

Notice SQL*Plus does the following:

1. Prompts you to enter a value for product_id_var.
2. Substitutes the value you entered for product_id_var in the WHERE clause.

SQL*Plus shows you the substitution in the old and new lines in the output, along with the line number in the query where the substitution was performed. In the previous example, you can see that the old and new lines indicate that product_id_var is set to 2 in the WHERE clause of the SELECT statement.

If you rerun the query using the slash character (/), SQL*Plus will ask you to enter a new value for product_id_var. For example:

```
SQL> /
Enter value for product_id_var: 3
old   3: WHERE product_id = &product_id_var
new   3: WHERE product_id = 3

PRODUCT_ID NAME                                PRICE
---------- ------------------------------ ----------
         3 Supernova                           25.99
```

Once again, SQL*Plus echoes the old line of the SQL statement (old 3: WHERE product_id = &product_id_var) followed by the new line containing the variable value you entered (new 3: WHERE product_id = 3).

Controlling Output Lines

You may control the output of the old and new lines using the SET VERIFY command. If you enter SET VERIFY OFF, the old and new lines are suppressed. For example:

```
SQL> SET VERIFY OFF
SQL> /
Enter value for product_id_var: 4

PRODUCT_ID NAME                              PRICE
---------- ------------------------------ ----------
         4 Tank War                          13.95
```

To turn the echoing of the lines back on, you enter SET VERIFY ON. For example:

```
SQL> SET VERIFY ON
```

Changing the Variable Definition Character

You can use the SET DEFINE command to specify a character other than ampersand (&) for defining a variable. The following example shows how to set the variable character to the pound character (#) using SET DEFINE and shows a new SELECT statement:

```
SQL> SET DEFINE '#'
SQL> SELECT product_id, name, price
  2  FROM products
  3  WHERE product_id = #product_id_var;
Enter value for product_id_var: 5
old   3: WHERE product_id = #product_id_var
new   3: WHERE product_id = 5

PRODUCT_ID NAME                              PRICE
---------- ------------------------------ ----------
         5 Z Files                           49.99
```

The next example uses SET DEFINE to change the character back to an ampersand:

```
SQL> SET DEFINE '&'
```

Substituting Table and Column Names Using Variables

You're not limited to using variables to substitute column values: you can also use variables to substitute the names of tables and columns. For example, the following query defines variables for you to enter a column name (col_var) or table name (table_var), as well as a column value (col_val_var):

```
SQL> SELECT name, &col_var
  2  FROM &table_var
  3  WHERE &col_var = &col_val;
Enter value for col_var: product_type_id
old   1: SELECT name, &col_var
new   1: SELECT name, product_type_id
```

```
Enter value for table_var: products
old   2: FROM &table_var
new   2: FROM products
Enter value for col_var: product_type_id
Enter value for col_val: 1
old   3: WHERE &col_var = &col_val
new   3: WHERE product_type_id = 1

NAME                           PRODUCT_TYPE_ID
------------------------------ ---------------
Modern Science                               1
Chemistry                                    1
```

You can avoid having to repeatedly enter a variable by using &&. For example:

```
SELECT name, &&col_var
FROM &table_name
WHERE &&col_var = &col_val;
```

Being able to supply column and table names, as well as variable values, gives you a lot of flexibility in writing interactive queries that another user may run. That user doesn't need to write the SQL: you can simply give them a script and have them enter the variable values for the query.

Defined Variables

You can define a variable prior to using that variable in a SQL statement. You may use these variables multiple times within a SQL statement. A defined variable persists until you explicitly remove it, redefine it, or exit SQL*Plus.

You define a variable using the `DEFINE` command. When you're finished with your variable, you remove it using `UNDEFINE`. You'll learn about each of these commands in this section. You'll also learn about the `ACCEPT` command, which allows you to define a variable and specify a data type for that variable.

You can also define variables in a SQL*Plus script and pass in values to those variables when you run the script. This enables you to write generic reports that any user can run—even if they're unfamiliar with SQL. You'll learn how to create simple reports in the section "Creating Simple Reports."

Defining and Listing Variables Using the DEFINE Command

You use the `DEFINE` command to both define a new variable and list the currently defined variables. The following example defines a variable named `product_id_var` and sets its value to 7:

```
SQL> DEFINE product_id_var = 7
```

You can view the definition of a variable using the `DEFINE` command followed by the name of the variable. The following example displays the definition of `product_id_var`:

```
SQL> DEFINE product_id_var
DEFINE PRODUCT_ID_VAR          = "7" (CHAR)
```

Notice that product_id_var is defined as a CHAR variable.

You can see all your session variables by entering DEFINE on its own line. For example:

```
SQL> DEFINE
...
DEFINE PRODUCT_ID_VAR  = "7" (CHAR)
```

You can use a defined variable to specify an element such as a column value in a SQL statement. For example, the following query uses the variable product_id_var defined earlier and references its value in the WHERE clause:

```
SQL> SELECT product_id, name, price
  2  FROM products
  3  WHERE product_id = &product_id_var;
old   3: WHERE product_id = &product_id_var
new   3: WHERE product_id = 7

PRODUCT_ID NAME                                PRICE
---------- ------------------------------ ----------
         7 Space Force 9                       13.49
```

Notice that you're not prompted to the value of product_id_var; that's because product_id_var was set to 7 when the variable was defined earlier.

Defining and Setting Variables Using the ACCEPT Command

The ACCEPT command waits for a user to enter a value for a variable. You can use the ACCEPT command to set an existing variable to a new value, or to define a new variable and initialize it with a value. The ACCEPT command also allows you to specify the data type for your variable. The simplified syntax for the ACCEPT command is as follows:

```
ACCEPT variable_name [type] [FORMAT format] [PROMPT prompt] [HIDE]
```

where

- *variable_name* specifies the name assigned to your variable.
- *type* specifies the data type for your variable. You can use the CHAR, NUMBER, and DATE types. By default, variables are defined using the CHAR type. DATE variables are actually stored as CHAR variables.
- *format* specifies the format used for your variable. Some examples include A15 (15 characters), 9999 (a four-digit number), and DD-MON-YYYY (a date). You can view the number formats in Table 3-4 of Chapter 3; you can view the date formats in Table 4-2 of Chapter 4.
- *prompt* specifies the text displayed by SQL*Plus as a prompt to the user to enter the variable's value.

- HIDE indicates the value entered for the variable is to be hidden. For example, you might want to hide passwords or other sensitive information. Hidden values are displayed using asterisks (*) as you enter the characters.

Let's take a look at some examples of the ACCEPT command. The first example defines a variable named customer_id_var as a two-digit number:

```
SQL> ACCEPT customer_id_var NUMBER FORMAT 99 PROMPT 'Customer id: '
Customer id: 5
```

The next example defines a DATE variable named date_var; the format for this DATE is DD-MON-YYYY:

```
SQL> ACCEPT date_var DATE FORMAT 'DD-MON-YYYY' PROMPT 'Date: '
Date: 12-DEC-2006
```

The next example defines a CHAR variable named password_var; the value entered is hidden using the HIDE option:

```
SQL> ACCEPT password_var CHAR PROMPT 'Password: ' HIDE
Password: *******
```

In Oracle9*i* and below, the value entered appears as a string of asterisk characters (*) to hide the value as you enter it. In Oracle10*i*, nothing is displayed as you type the value.

You can view your variables using the DEFINE command. For example:

```
SQL> DEFINE
...
DEFINE CUSTOMER_ID_VAR =          5 (NUMBER)
DEFINE DATE_VAR        = "12-DEC-2006" (CHAR)
DEFINE PASSWORD_VAR    = "1234567" (CHAR)
```

Notice that date_var is stored as a CHAR.

Removing Variables Using the UNDEFINE Command

You remove variables using the UNDEFINE command. The following example uses UNDEFINE to remove customer_id_var, date_var, and password_var:

```
SQL> UNDEFINE customer_id_var
SQL> UNDEFINE date_var
SQL> UNDEFINE password_var
```

NOTE
*All your variables are removed when you exit SQL*Plus, even if you don't explicitly remove them using the UNDEFINE command.*

Creating Simple Reports

You can use temporary and defined variables in a SQL*Plus script. This allows you to create scripts that prompt the user for entry of variables that can then be used to generate reports. You'll find the SQL*Plus scripts referenced in this section in the Zip file you can download from this book's web site.

TIP
*Bear in mind that SQL*Plus was not specifically designed as a reporting tool. If you have complex reporting requirements, you should use software like Oracle Reports.*

Using Temporary Variables in a Script

The following script `report1.sql` uses a temporary variable named `product_id_var` in the `WHERE` clause of a `SELECT` statement:

```
-- suppress display of the statements and verification messages
SET ECHO OFF
SET VERIFY OFF

SELECT product_id, name, price
FROM products
WHERE product_id = &product_id_var;
```

The `SET ECHO OFF` command stops SQL*Plus from displaying the SQL statements and commands in the script. `SET VERIFY OFF` suppresses display of the verification messages. I put these two commands in to minimize the number of extra lines displayed by SQL*Plus when you run the script.

You can run `report1.sql` in SQL*Plus using the `@` command. For example:

```
SQL> @ C:\SQL\report1.sql
Enter value for product_id_var: 2

PRODUCT_ID NAME                                PRICE
---------- ------------------------------ ----------
         2 Chemistry                              30
```

You can give this script to another user and they can run it without them having to know SQL.

Using Defined Variables in a Script

The following script (named `report2.sql`) uses the `ACCEPT` command to define a variable named `product_id_var`:

```
SET ECHO OFF
SET VERIFY OFF

ACCEPT product_id_var NUMBER FORMAT 99 PROMPT 'Enter product id: '
```

```
SELECT product_id, name, price
FROM products
WHERE product_id = &product_id_var;

-- clean up
UNDEFINE product_id_var
```

Notice that a user-friendly prompt is specified for the entry of `product_id_var`, and that `product_id_var` is removed at the end of the script—this makes the script cleaner.

You can run the `report2.sql` script using SQL*Plus:

```
SQL> @ C:\SQL\report2.sql
Enter product id: 4

PRODUCT_ID NAME                           PRICE
---------- ------------------------------ ----------
         4 Tank War                       13.95
```

Passing a Value to a Variable in a Script

You can pass a value to a variable when you run your script. When you do this, you reference the variable in your script using a number. The following script `report3.sql` shows an example of this; notice the variable is identified using `&1`:

```
SET ECHO OFF
SET VERIFY OFF

SELECT product_id, name, price
FROM products
WHERE product_id = &1;
```

When you run `report3.sql`, you supply the variable's value after the script name. The following example passes the value 3 to `report3.sql`:

```
SQL> @ C:\SQL\report3.sql 3
PRODUCT_ID NAME                           PRICE
---------- ------------------------------ ----------
         3 Supernova                      25.99
```

You can add any number of parameters, with each value specified on the command line corresponding to the matching number in the file. The first parameter corresponds to `&1`, the second to `&2`, and so on. The following script `report4.sql` shows an example of this:

```
SET ECHO OFF
SET VERIFY OFF

SELECT product_id, product_type_id, name, price
FROM products
```

```
WHERE product_type_id = &1
AND price > &2;
```

The following example run of `report4.sql` shows the addition of two values for `&1` and `&2`, which are set to 1 and 9.99, respectively:

```
SQL> @ C:\SQL\report4.sql 1 9.99

PRODUCT_ID PRODUCT_TYPE_ID NAME                                PRICE
---------- --------------- ------------------------------ ----------
         1               1 Modern Science                      19.95
         2               1 Chemistry                              30
```

Because `&1` is set to 1, the `product_type_id` column in the `WHERE` clause is set to 1. Also, `&2` is set to 9.99, so the price column in the `WHERE` clause is set to 9.99. Therefore, rows with a `product_type_id` of 1 and a `price` greater than 9.99 are displayed.

Adding a Header and Footer

You add a header and footer to your report using the `TTITLE` and `BTITLE` commands. The following script `report5.sql` shows this:

```
TTITLE 'Product Report'
BTITLE 'Thanks for running the report'

SET ECHO OFF
SET VERIFY OFF
SET PAGESIZE 30
SET LINESIZE 70
CLEAR COLUMNS
COLUMN product_id HEADING ID FORMAT 99
COLUMN name HEADING 'Product Name' FORMAT A20 WORD_WRAPPED
COLUMN description HEADING Description FORMAT A30 WORD_WRAPPED
COLUMN price HEADING Price FORMAT $99.99

SELECT product_id, name, description, price
FROM products;

CLEAR COLUMNS
```

The following example shows a run of `report5.sql`:

```
SQL> @ C:\SQL\report5.sql

Fri May 16                                                  page    1
                            Product Report

 ID Product Name         Description                      Price
--- -------------------- ------------------------------ -------
```

```
  1 Modern Science       A description of modern        $19.95
                         science

  2 Chemistry            Introduction to Chemistry      $30.00
  3 Supernova            A star explodes                $25.99
  4 Tank War             Action movie about a future    $13.95
                         war

  5 Z Files              Series on mysterious           $49.99
                         activities

  6 2412: The Return     Aliens return                  $14.95
  7 Space Force 9        Adventures of heroes           $13.49
  8 From Another Planet  Alien from another planet      $12.99
                         lands on Earth

  9 Classical Music      The best classical music       $10.99
 10 Pop 3                The best popular music         $15.99
 11 Creative Yell        Debut album                    $14.99
 12 My Front Line        Their greatest hits            $13.49

                    Thanks for running the report
```

Computing Subtotals

You can add a subtotal for a column using a combination of the BREAK ON and COMPUTE commands. BREAK ON causes SQL*Plus to break up output based on a change in a column value, and COMPUTE causes SQL*Plus to compute a value for a column.

The following script report6.sql shows how to compute a subtotal for products of the same type:

```
BREAK ON product_type_id
COMPUTE SUM OF price ON product_type_id

SET ECHO OFF
SET VERIFY OFF
SET PAGESIZE 50
SET LINESIZE 70

CLEAR COLUMNS
COLUMN price HEADING Price FORMAT $999.99

SELECT product_type_id, name, price
FROM products
ORDER BY product_type_id;

CLEAR COLUMNS
```

The following example shows a run of report6.sql:

```
SQL> @ C:\SQL\report6.sql

PRODUCT_TYPE_ID NAME                           Price
--------------- ------------------------------ --------
              1 Modern Science                 $19.95
                Chemistry                      $30.00
***************                                --------
sum                                            $49.95
              2 Supernova                      $25.99
                Tank War                       $13.95
                Z Files                        $49.99
                2412: The Return               $14.95
***************                                --------
sum                                            $104.88
              3 Space Force 9                  $13.49
                From Another Planet            $12.99
***************                                --------
sum                                            $26.48
              4 Classical Music                $10.99
                Pop 3                          $15.99
                Creative Yell                  $14.99
***************                                --------
sum                                            $41.97
                My Front Line                  $13.49
***************                                --------
sum                                            $13.49
```

Notice that whenever a new value for product_type_id is encountered, SQL*Plus breaks up the output and computes a sum for the price columns for the rows with the same product_type_id. The product_type_id value is only shown once for rows with the same product_type_id. For example, Modern Science and Chemistry are both books and have a product_type_id of 1, and 1 is shown once for Modern Science. The sum of the prices for these two books is $49.95. The other sections of the report contain the sum of the prices for products with different product_type_id values.

Automatically Generating SQL Statements

In this last section, I'll briefly show you a technique of writing SQL statements that produce other SQL statements. This is very useful and can save you a lot of typing when writing SQL statements that are similar. One simple example is a SQL statement that produces DROP TABLE statements that remove tables from a database. The following query produces a series of DROP TABLE statements that drop the tables in the store schema:

```
SELECT 'DROP TABLE ' || table_name || ';'
FROM user_tables;

'DROPTABLE'||TABLE_NAME||';'
------------------------------------------
```

```
DROP TABLE COUPONS;
DROP TABLE CUSTOMERS;
DROP TABLE EMPLOYEES;
DROP TABLE PRODUCTS;
DROP TABLE PRODUCT_TYPES;
DROP TABLE PROMOTIONS;
DROP TABLE PURCHASES;
DROP TABLE PURCHASES_TIMESTAMP_WITH_TZ;
DROP TABLE PURCHASES_WITH_LOCAL_TZ;
DROP TABLE PURCHASES_WITH_TIMESTAMP;
DROP TABLE SALARY_GRADES;
```

NOTE
`user_tables` contains the details of the tables in the user's schema. The `table_name` column contains names of the tables.

You can spool the generated SQL statements to a file and use them later.

Summary

In this chapter, you learned how to

- View the structure of a table
- Edit a SQL statement
- Save, retrieve, and run files containing SQL and SQL*Plus commands
- Format column output and set the page and line sizes
- Use variables in SQL*Plus
- Create simple reports
- Write SQL statements that generate other SQL statements

In the next chapter, you'll learn how to nest one query within another. The nested query is known as a subquery.