
Creating and Running PL/SQL Code

Even if they never give a second thought to tasks such as system design or unit testing, all PL/SQL programmers must be familiar with some basic operational tasks:

- Navigating the database
- Creating and editing PL/SQL source code
- Compiling the PL/SQL source code, and correcting any code errors (and, optionally, warnings) noted by the compiler
- Executing the compiled program from some environment
- Examining the results of program execution (screen output, changes to tables, etc.)

Unlike standalone languages such as C, PL/SQL is hosted inside an Oracle execution environment (it is an *embedded language*), so there are some unexpected nuances to all of these tasks: some are pleasant surprises; others, consternations. This chapter will show you how to accomplish these tasks at the most basic level (using SQL*Plus), with a brief tour of the nuances sprinkled in. It concludes with some drive-by examples of making calls to PL/SQL from inside several common programming environments such as PHP and C. For more detailed information about compilation and other more advanced tasks, see [Chapter 20](#).

Navigating the Database

Everybody who chooses to write PL/SQL programs does so to work with the contents of an Oracle database. It is, therefore, no surprise that you will need to know how to “get around” the Oracle database where your code is going to run. You will want to examine the data structures (tables, columns, sequences, user-defined types, etc.) in the database, as well as the signatures of any existing stored programs you will be invoking.

You will probably also need to know something about the actual contents (columns, constraints, etc.) of the tables.

There are two distinct approaches you can take to database navigation:

1. Use an IDE (integrated development environment, a fancy name for a fancy editor) like Toad, SQL Developer, PL/SQL Developer, or SQL Navigator. They all offer visual browsers that support point-and-click navigation.
2. Run scripts in a command-line environment like SQL*Plus, which queries the contents of data dictionary views like `ALL_OBJECTS` or `USER_OBJECTS` (demonstrated later in this chapter).

I strongly recommend that you use a graphical IDE. If you have been around Oracle long enough, you might be addicted to and fairly productive with your scripts. For the rest of us, a graphical interface is much easier to work with and understand—and much more productive—than scripts.

Chapter 20 also offers examples of using several data dictionary views for working with your PL/SQL code base.

Creating and Editing Source Code

These days, programmers have many, many choices for code editors, from the simplest text editor to the most exotic development environments. And they do make very different choices. One of the authors of this book, Steven Feuerstein, is rather addicted to the Toad IDE. He is a very typical user, familiar with perhaps only 10% of all the functionality and buttons, but relying heavily on those features. Bill Pribyl, on the other hand, describes himself as “something of an oddball in that I like to use a fairly plain text editor to write PL/SQL programs. My one concession is that it automatically indents code as I type, and it displays keywords, comments, literals, and variables in different colors.”

The most sophisticated editors will do much more than indentation and keyword coloring; they also offer graphical debuggers, perform keyword completion, preview subprograms of packages as you type their names, display subprogram parameters, and highlight the specific row and column where the compiler reported an error. Some editors also have “hyperlinking” features that allow you to quickly browse to the declaration of a variable or subprogram. But the need for most of these features is common across many compiled languages.

What is unique about PL/SQL is the fact that the source code for stored programs must be loaded into the database before it can be compiled and executed. This in-database copy can usually be retrieved by a programmer who has sufficient permissions. We can immediately recognize a host of code management issues, including:

- How and where does a programmer find the “original” copy of a stored program?
- Does it live on disk or does it just live in the database?
- How and how often do we perform backups?
- How do we manage multi-developer access to the code? That is, do we use a software version control system?

These questions should be answered before you begin development of an application, preferably by making choices about which software tools will do this work for you. While there is no single set of tools or processes that work best for all development teams, I can tell you that I always store the “original” source code in files—I strongly suggest that you *not* use the RDBMS as your code repository.

In the next section I will demonstrate how you can use SQL*Plus to accomplish many basic tasks for PL/SQL development. These same tasks can be completed in your IDE.

SQL*Plus

The granddaddy of Oracle frontends, Oracle’s SQL*Plus provides a *command-line interpreter* for both SQL and PL/SQL. That is, it accepts statements from the user, sends them off to the Oracle server, and displays the results.

Often maligned for its user interface, SQL*Plus is one of my favorite Oracle tools. I actually *like* the lack of fancy gizmos and menus. Ironically, when I started using Oracle (circa 1986), this product’s predecessor was boldly named UFI—*User-Friendly Interface*. Two decades later, even the latest version of SQL*Plus is still unlikely to win any user friendliness awards, but at least it doesn’t crash very often.

Oracle has, over the years, offered different versions of SQL*Plus, including:

As a console program

This is a program that runs from a shell or command prompt (an environment that is sometimes called a *console*).¹

As a pseudo-GUI program

This form of SQL*Plus is available only on Microsoft Windows. I call it a “pseudo-GUI” because it looks pretty much like the console program but with bitmapped fonts; few other features distinguish it from the console program. Beware: Oracle has been threatening to desupport this product for years, and it hasn’t really been updated since Oracle8i Database.

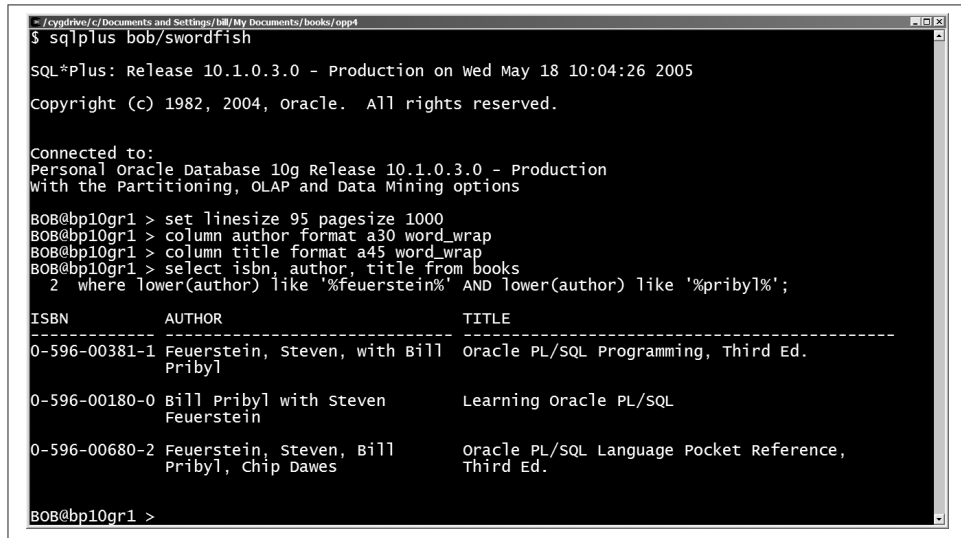
1. Oracle calls this the “command-line interface” version of SQL*Plus, but I find that somewhat confusing, because two of the three styles provide a command-line interface.

Via iSQL*Plus

This program executes from a web browser connected to a middle-tier machine running Oracle's HTTP server and iSQL*Plus server.

Starting with Oracle Database 11g, Oracle ships only the console program (*sqlplus.exe*).

Figure 2-1 is a screenshot of a SQL*Plus console-style session.



```
~/cygdrive/c:/Documents and Settings/bill/My Documents/books/opp4
$ sqlplus bob/swordfish

SQL*Plus: Release 10.1.0.3.0 - Production on Wed May 18 10:04:26 2005
copyright (c) 1982, 2004, Oracle. All rights reserved.

Connected to:
Personal Oracle Database 10g Release 10.1.0.3.0 - Production
With the Partitioning, OLAP and Data Mining options

BOB@bp10gr1 > set linesize 95 pagesize 1000
BOB@bp10gr1 > column author format a30 word_wrap
BOB@bp10gr1 > column title format a45 word_wrap
BOB@bp10gr1 > select isbn, author, title from books
2 where lower(author) like '%feuerstein%' AND lower(author) like '%pribyl%';

ISBN                AUTHOR                                TITLE
-----
0-596-00381-1  Feuerstein, Steven, with Bill Pribyl  Oracle PL/SQL Programming, Third Ed.
0-596-00180-0  Bill Pribyl with Steven Feuerstein    Learning Oracle PL/SQL
0-596-00680-2  Feuerstein, Steven, Bill Pribyl, Chip Dawes  Oracle PL/SQL Language Pocket Reference, Third Ed.

BOB@bp10gr1 >
```

*Figure 2-1. SQL*Plus in a console session*

Usually, I prefer the console program because:

- It tends to draw the screen faster, which can be significant for queries with lots of output.
- It has a more complete command-line history (on Microsoft Windows platforms, at least).
- It has a much easier way of changing visual characteristics such as font, color, and scroll buffer size.
- It is available virtually everywhere that Oracle server or client tools are installed.

Starting Up SQL*Plus

To start the console version of SQL*Plus, you can simply type “sqlplus” at the operating system prompt (designated by “OS>”):

```
OS> sqlplus
```

This works for both Unix-based and Microsoft operating systems. SQL*Plus should display a startup banner and then prompt you for a username and password:

```
SQL*Plus: Release 11.1.0.6.0 - Production on Fri Nov 7 10:28:26 2008
```

```
Copyright (c) 1982, 2007, Oracle. All rights reserved.
```

```
Enter user-name: bob
```

```
Enter password: swordfish
```

```
Connected to:
```

```
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - 64bit
```

```
SQL>
```

Seeing the “SQL>” prompt is your cue that your installation is set up properly. (The password won’t echo on the screen.)

You can also launch SQL*Plus with the username and password on the command line:

```
OS> sqlplus bob/swordfish
```

I do *not* recommend this, because some operating systems provide a way for other users to see your command-line arguments, which would allow them to read your password. On multiuser systems, you can instead use the /NOLOG option to start SQL*Plus without connecting to the database, and then supply the username and password via the CONNECT command:

```
OS> sqlplus /nolog
```

```
SQL*Plus: Release 11.1.0.6.0 - Production on Fri Nov 7 10:28:26 2008
```

```
Copyright (c) 1982, 2007, Oracle. All rights reserved.
```

```
SQL> CONNECT bob/swordfish
```

```
SQL> Connected.
```

If the computer you’re running SQL*Plus on also has a properly configured Oracle Net² installation, *and* you have been authorized by the database administrator to connect to remote databases (that is, database servers running on other computers), you can connect to these other databases from SQL*Plus. Doing so requires knowing an Oracle Net *connect identifier* (also known as a *service name*) that you must supply along with your username and password. A connect identifier could look like this:

```
hqhr.WORLD
```

To use this identifier, you can append it to your username and password, separated by an at sign (@):

2. Oracle Net is the current name for the product previously known as Net8 and SQL*Net.

```
SQL> CONNECT bob/swordfish@hqhr.WORLD
SQL> Connected.
```

When starting the pseudo-GUI version of SQL*Plus, supplying your credentials is straightforward, although it calls the connect identifier a *host string* (see [Figure 2-2](#)). If you want to connect to a database server running on the local machine, just leave the Host String field blank.

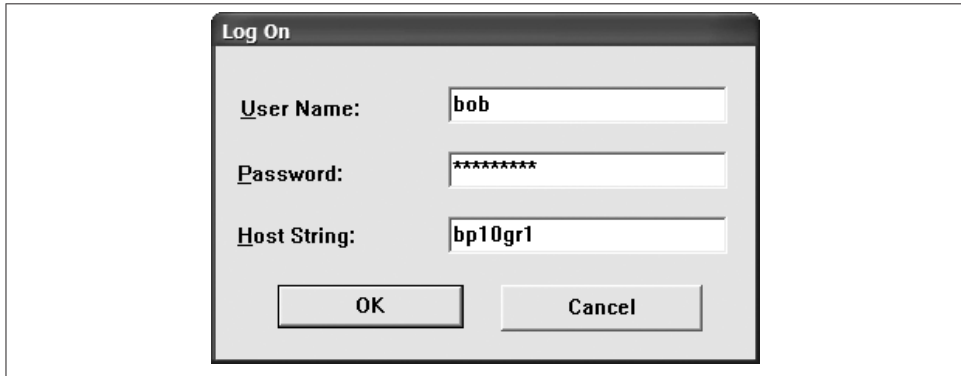


Figure 2-2. The GUI login screen of SQL*Plus

Once you have SQL*Plus running, you can do all kinds of things. Here are some of the most common:

- Run a SQL statement.
- Compile and store a PL/SQL program in the database.
- Run a PL/SQL program.
- Issue a SQL*Plus-specific command.
- Run a script that contains a mix of the preceding.

We'll take a look at these in the following sections.

Running a SQL Statement

The default terminator in SQL*Plus for SQL statements is the semicolon, but you can change that terminator character.

In the console version of SQL*Plus, the query:

```
SELECT isbn, author, title FROM books;
```

produces output similar to that shown in [Figure 2-1](#).³

Running a PL/SQL Program

So, here we go (drumroll, please). Let's type a short PL/SQL program into SQL*Plus:

```
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE('Hey look, ma!');
  3 END;
  4 /
```

PL/SQL procedure successfully completed.

SQL>

Oops. Although it has successfully completed, this particular program was supposed to invoke PL/SQL's built-in program that echoes back some text. SQL*Plus's somewhat annoying behavior is to suppress such output by default. To get it to display properly, you must use a SQL*Plus command to turn on SERVEROUTPUT:

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE('Hey look, Ma!');
  3 END;
  4 /
Hey look, Ma!
```

PL/SQL procedure successfully completed.

SQL>

I generally put the SERVEROUTPUT command in my startup file (see [“Loading your own custom environment automatically on startup” on page 35](#)), causing it to be enabled until one of the following occurs:

- You disconnect, log off, or otherwise end your session.
- You explicitly set SERVEROUTPUT to OFF.
- The Oracle database discards session state either at your request or because of a compilation error (see [“Recompiling Invalid Program Units” on page 773](#)).

3. Well, I cheated a bit in that figure because I used some column formatting commands. If this were a book about SQL*Plus or how to display database data, I would expound on the many ways SQL*Plus lets you control the appearance of the output by setting various formatting and display preferences. You can take my word for it, though: there are more options than you can shake a stick at.

- In Oracle versions through Oracle9i Database Release 2, you issue a new CONNECT statement; in subsequent versions, SQL*Plus automatically reruns your startup file after each CONNECT.

When you enter SQL or PL/SQL statements into the console or pseudo-GUI SQL*Plus, the program assigns a number to each line after the first. There are two benefits to the line numbers: first, they help you designate which line to edit with the built-in line editor (which you might actually use one day); and second, if the database detects an error in your code, it will usually report the error accompanied by a line number. You'll have plenty of opportunities to see *that* behavior in action.

To tell SQL*Plus that you're done entering a PL/SQL statement, you must usually include a trailing slash (see line 4 in the previous example). Although mostly harmless, the slash has several important characteristics:

- The meaning of the slash is “execute the most recently entered statement,” regardless of whether the statement is SQL or PL/SQL.
- The slash is a command unique to SQL*Plus; it is *not* part of the PL/SQL language, nor is it part of SQL.
- It must appear on a line by itself; no other commands can be included on the line.
- In most versions of SQL*Plus prior to Oracle9i Database, if you accidentally precede the slash with any spaces, it doesn't work! Beginning with Oracle9i Database, SQL*Plus conveniently overlooks leading whitespace. Trailing space doesn't matter in any version.

As a convenience feature, SQL*Plus offers PL/SQL users an EXECUTE command, which saves typing the BEGIN, END, and trailing slash. So, the following is equivalent to the short program I ran earlier:

```
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE('Hey look, Ma!')
```

A trailing semicolon is optional, but I prefer to omit it. As with most SQL*Plus commands, EXECUTE can be abbreviated and is case insensitive, so most interactive use gets reduced to:

```
SQL> EXEC dbms_output.put_line('Hey look, Ma!')
```


Running a Script

Almost any statement that works interactively in SQL*Plus can be stored in a file for repeated execution. The easiest way to run such a script is to use the SQL*Plus `@` command.⁴ For example, this runs all the commands in the file *abc.pkg*:

```
SQL> @abc.pkg
```

The file must live in my current directory (or on SQLPATH somewhere).

If you prefer words to at signs, you can use the equivalent `START` command:

```
SQL> START abc.pkg
```

and you will get identical results. Either way, this command causes SQL*Plus to do the following:

1. Open the file named *abc.pkg*.
2. Sequentially attempt to execute all of the SQL, PL/SQL, and SQL*Plus statements in the file.
3. When complete, close the file and return you to the SQL*Plus prompt (unless the file invokes the `EXIT` statement, which will cause SQL*Plus to quit).

For example:

```
SQL> @abc.pkg
```

```
Package created.
```

```
Package body created.
```

```
SQL>
```

The default behavior is to display only the output from the individual statements on the screen; if you want to see the original source from the file, use the SQL*Plus command `SET ECHO ON`.

In my example, I've used a filename extension of *.pkg*. If I leave off the extension, this is what happens:

```
SQL> @abc
```

```
SP2-0310: unable to open file "abc.sql"
```

As you can see, the default file extension is *.sql*. By the way, the “SP2-0310” is the Oracle-supplied error number, and “SP2” means that it is unique to SQL*Plus. (For more details

4. `START`, `@`, and `@@` commands are available in the nonbrowser versions of SQL*Plus. In *iSQL*Plus*, you can use the “Browse” and “Load Script” buttons for a similar result.

about SQL*Plus error messages, refer to Oracle's *SQL*Plus User's Guide and Reference*.)

What Is the "Current Directory"?

Any time you launch SQL*Plus from an operating system command prompt, SQL*Plus treats the operating system's then-current directory as its own current directory. In other words, if I were to start up using:

```
C:\BOB\FILES> sqlplus
```

then any file operations inside SQL*Plus (such as opening or running a script) would default to the directory *C:\BOB\FILES*.

If you use a shortcut or menu option to launch SQL*Plus, the current directory is the directory the operating system associates with the launch mechanism. So how would you change the current directory once you're inside SQL*Plus? It depends on the version. In the console program, you can't do it. You have to exit, change directories in the operating system, and restart SQL*Plus. In the GUI version, though, completing a File→Open or File→Save menu command will have the side effect of changing the current directory.

If your script file is in another directory, you can precede the filename with the path:⁵

```
SQL> @/files/src/release/1.0/abc.pkg
```

The idea of running scripts in other directories raises an interesting question. What if *abc.pkg* is located in this other directory and, in turn, calls other scripts? It might contain the lines:

```
REM  Filename: abc.pkg
@abc.pks
@abc.pkb
```

(Any line beginning with REM is a comment or "remark" that SQL*Plus ignores.) Executing the *abc.pkg* script is supposed to run *abc.pks* and *abc.pkb*. But because I have not included path information, where will SQL*Plus look for these other files? Let's see:

```
C:\BOB\FILES> sqlplus
...
SQL> @/files/src/release/1.0/abc.pkg
SP2-0310: unable to open file "abc.pks"
SP2-0310: unable to open file "abc.pkb"
```

It looks only in the directory where I started.

5. You can use forward slashes as directory delimiters on both Unix/Linux and Microsoft operating systems. This allows your scripts to port more easily between operating systems.

To address this problem, Oracle created the @@ command. This double at sign means during this call, “pretend I have changed the current directory to be that of the currently executing file.” So, the preferred way of writing the calls in the *abc.pkg* script is:

```
REM  Filename: abc.pkg
@@abc.pks
@@abc.pkb
```

Now I get:

```
C:\BOB\FILES> sqlplus
...
SQL> @/files/src/release/1.0/abc.pkg
```

Package created.

Package body created.

...just as I was hoping.

Other SQL*Plus Tasks

There are dozens of commands specific to SQL*Plus, but I have space to mention only a few more that are particularly important or particularly confusing. For a thorough treatment of this venerable product, get a copy of Jonathan Gennick’s book *Oracle SQL*Plus: The Definitive Guide*, or for quick reference, his *Oracle SQL*Plus Pocket Reference*.

Setting your preferences

You can change the behavior of SQL*Plus, as you can with many command-line environments, by changing the value of some of its built-in variables and settings. You have already seen one example, the SET SERVEROUTPUT statement. There are many variations on the SQL*Plus SET command, such as SET SUFFIX (changes the default file extension) and SET LINESIZE *n* (sets the maximum number of characters in each displayed line before wrapping). To see all the SET values applicable to your current session, use the command:

```
SQL> SHOW ALL
```

SQL*Plus can also create and manipulate its own in-memory variables, and it sets aside a few special variables that will affect its behavior. Actually, there are two separate types of variables in SQL*Plus: DEFINES and bind variables. To assign a value to a DEFINE variable, you can use the DEFINE command:

```
SQL> DEFINE x = "the answer is 42"
```

To view the value of x, specify:

```
SQL> DEFINE x
DEFINE X = "the answer is 42" (CHAR)
```

You would refer to such a variable using an ampersand (&). SQL*Plus does a simple substitution before sending the statement to the Oracle database, so you will need single-quote marks around the variable when you want to use it as a literal string:

```
SELECT '&x' FROM DUAL;
```

For bind variables, you first declare the variable. You can then use it in PL/SQL, and display it using the SQL*Plus PRINT command:

```
SQL> VARIABLE x VARCHAR2(10)
SQL> BEGIN
2     :x := 'hullo';
3 END;
4 /
```

PL/SQL procedure successfully completed.

```
SQL> PRINT :x
```

```
X
-----
hullo
```

This can get a little bit confusing because there are now two different “x” variables, one that has been defined and one that has been declared:

```
SQL> SELECT :x, '&x' FROM DUAL;
old 1: SELECT :x, '&x' FROM DUAL
new 1: SELECT :x, 'the answer is 42' FROM DUAL

:X                                'THEANSWERIS42'
-----
hullo                             the answer is 42
```

Just remember that DEFINES are always character strings expanded by SQL*Plus, and declared variables are used as true bind variables in SQL and PL/SQL.

Saving output to a file

Frequently, you will want to save output from a SQL*Plus session to a file—perhaps because you are generating a report, or because you want a record of your actions, or because you are dynamically generating commands to execute later. An easy way to do this in SQL*Plus is to use its SPOOL command:

```
SQL> SPOOL report
SQL> @run_report
```

...output scrolls past and gets written to the file report.lst...

```
SQL> SPOOL OFF
```

The first command, SPOOL report, tells SQL*Plus to save everything from that point forward into the file *report.lst*. The file extension of *.lst* is the default, but you can override it by supplying your own extension in the SPOOL command:

```
SQL> SPOOL report.txt
```

SPOOL OFF tells SQL*Plus to stop saving the output and to close the file.

Exiting SQL*Plus

To exit SQL*Plus and return to the operating system, use the EXIT command:

```
SQL> EXIT
```

If you happen to be spooling when you exit, SQL*Plus will stop spooling and close the spool file.

What happens if you modify some table data during your session but then exit before ending the transaction with an explicit transaction control statement? By default, exiting SQL*Plus forces a COMMIT, unless your sessions end with a SQL error and you have issued the SQL*Plus WHENEVER SQLERROR EXIT ROLLBACK command (see the section [“Error Handling in SQL*Plus” on page 36](#)).

To disconnect from the database but remain connected to SQL*Plus, use the command DISCONNECT, which will look something like this in action:

```
SQL> DISCONNECT
Disconnected from Personal Oracle Database 10g Release 10.1.0.3.0 - Production
With the Partitioning, OLAP and Data Mining options
SQL>
```

You don't have to use DISCONNECT to change connections—you can just issue a CONNECT instead, and SQL*Plus will drop the first connection before connecting you to the new one. However, there is a good reason why you might want to disconnect before reconnecting: if you happen to be using operating system authentication,⁶ the script might reconnect itself automatically... maybe to the wrong account. I've seen it happen.

Editing a statement

SQL*Plus keeps the most recently issued statement in a buffer, and you can edit this statement using either the built-in line editor or an external editor of your choosing. To start with, I'll show how to set and use an external editor.

6. Operating system authentication is a way that you can bypass the username/password prompt when you log into SQL*Plus.

Use the EDIT command to have SQL*Plus save the current command buffer to a file, temporarily pause SQL*Plus, and invoke the editor:

```
SQL> EDIT
```

By default, the file will be saved with the name *afiedt.buf*, but you can change that with the SET EDITFILE command. Or, if you want to edit an existing file, just supply its name as an argument to EDIT:

```
SQL> EDIT abc.pkg
```

Once you've saved the file and exited the editor, the SQL*Plus session will read the contents of the newly edited file into its buffer, and then resume.

The default external editors that Oracle assumes are:

- ed for Unix, Linux, and relatives
- Notepad for Microsoft Windows variants

Although the selection of default editors is actually hardcoded into the *sqlplus* executable file, you can easily change the current editor by assigning your own value to the SQL*Plus `_EDITOR` variable. Here's an example that I frequently use:

```
SQL> DEFINE _EDITOR = /bin/vi
```

where */bin/vi* is the full path to an editor that's popular among a handful of strange people. I recommend using the editor's full pathname here, for security reasons.

If you really want to use the SQL*Plus built-in line editor (and it *can* be really handy), the essential commands you need to know are:

L

Lists the most recent statement.

n

Makes the *n*th line of the statement the current line.

DEL

Deletes the current line.

C /old/new/

In the current line, changes the first occurrence of *old* to *new*. The delimiter (here a forward slash) can be any arbitrary character.

n text

Makes *text* the current text of line *n*.

I

Inserts a line below the current line. To insert a new line prior to line 1, use a line zero command (e.g., 0 *text*).

Loading your own custom environment automatically on startup

To customize your SQL*Plus environment and have it assign your preferences from one session to the next, you will want to edit one or both of its autostartup scripts. The way SQL*Plus behaves on startup is:

1. It searches for the file `$ORACLE_HOME/sqlplus/admin/glogin.sql` and, if found, executes any commands it contains. This “global” login script applies to everyone who executes SQL*Plus from that Oracle home, no matter which directory they start in.
2. Next, it runs the file `login.sql` in the current directory, if it exists.⁷

The startup script can contain the same kinds of statements as any other SQL*Plus script: SET commands, SQL statements, column formatting commands, and the like.

Neither file is required to be present. If both files are present, `glogin.sql` executes, followed by `login.sql`; in the case of conflicting preferences or variables, the last setting wins.

Here are a few of my favorite `login.sql` settings:

```
REM Number of lines of SELECT statement output before reprinting headers
SET PAGESIZE 999
```

```
REM Width of displayed page, expressed in characters
SET LINESIZE 132
```

```
REM Enable display of DBMS_OUTPUT messages. Use 1000000 rather than
REM "UNLIMITED" for databases earlier than Oracle Database 10g Release 2
SET SERVEROUTPUT ON SIZE UNLIMITED FORMAT WRAPPED
```

```
REM Change default to "vi improved" editor
DEFINE _EDITOR = /usr/local/bin/vim
```

```
REM Format misc columns commonly retrieved from data dictionary
COLUMN segment_name FORMAT A30 WORD_WRAP
COLUMN object_name FORMAT A30 WORD_WRAP
```

```
REM Set the prompt (works in SQL*Plus
REM in Oracle9i Database or later)
SET SQLPROMPT "_USER'@'_CONNECT_IDENTIFIER > "
```

7. If it doesn't exist, and you have set the environment variable `SQLPATH` to one or more colon-delimited directories, SQL*Plus will search through those directories one at a time and execute the first `login.sql` that it finds. As a rule, I don't use `SQLPATH` because I am easily confused by this sort of skulking about.

Error Handling in SQL*Plus

The way SQL*Plus communicates success depends on the class of command you are running. With most SQL*Plus-specific commands, you can calibrate success by the absence of an error message. Successful SQL and PL/SQL commands, on the other hand, usually result in some kind of positive textual feedback.

If SQL*Plus encounters an error in a SQL or PL/SQL statement, it will, by default, report the error and continue processing. This behavior is desirable when you're working interactively. But when you're executing a script, there are many cases in which you'll want an error to cause SQL*Plus to terminate. Use the following command to make that happen:

```
SQL> WHENEVER SQLERROR EXIT SQL.SQLCODE
```

Thereafter in the current session, SQL*Plus terminates if the database server returns any error messages in response to a SQL or PL/SQL statement. The `SQL.SQLCODE` part means that, when SQL*Plus terminates, it sets its return code to a nonzero value, which you can detect in the calling environment.⁸ Otherwise, SQL*Plus always ends with a 0 return code, which may falsely imply that the script succeeded.

Another form of this command is:

```
SQL> WHENEVER SQLERROR EXIT SQL.SQLCODE ROLLBACK
```

which means that you also want SQL*Plus to roll back any uncommitted changes prior to exiting.

Why You Will Love and Hate SQL*Plus

In addition to the features you just read about, the following are some particular features of SQL*Plus that you will come to know and love:

- With SQL*Plus, you can run “batch” programs, supplying application-specific arguments on the *sqlplus* command line and referring to them in the script using `&1` (first argument), `&2` (second argument), etc.
- SQL*Plus provides complete and up-to-date support for all SQL and PL/SQL statements. This can be important when you're using features unique to Oracle. Third-party environments may not provide 100% coverage; for example, some have been slow to add support for Oracle's object types, which were introduced a number of years ago.

8. Using, for example, `$?` in the Unix shell or `%ERRORLEVEL%` in Microsoft Windows.

- SQL*Plus runs on all of the same hardware and operating system platforms on which the Oracle server runs.

But as with any tool, there are going to be some irritations too:

- In console versions of SQL*Plus, the statement buffer is limited to the most recently used statement; SQL*Plus offers no further command history.
- With SQL*Plus, there are no modern command-interpreter features such as automatic completion of keywords or hints about which database objects are available while you are typing in a statement.
- Online help consists of minimal documentation of the SQL*Plus command set. (Use *HELP command* to get help on a specific command.)
- There is no ability to change the current directory once you've started SQL*Plus. This can be annoying when opening or saving scripts if you don't like typing full pathnames. If you discover that you're in an inconvenient directory, you have to quit SQL*Plus, change directories, and restart SQL*Plus.
- Unless I break down and use what I consider the dangerous *SQLPATH* feature, SQL*Plus looks only in the startup directory for *login.sql*; it would be better if it would fall back to look in my home directory for the startup script.

The bottom line is that SQL*Plus is something of a “real programmer’s” tool that is neither warm nor fuzzy. But it is ubiquitous, doesn't crash, and is likely to be supported as long as there is an Oracle Corporation.

Performing Essential PL/SQL Tasks

Let's turn to the highlights of creating, running, deleting, and otherwise managing PL/SQL programs, using SQL*Plus as the frontend. Don't expect to be overwhelmed with detail here; treat this section as a glimpse of topics that will be covered in much greater detail in the chapters ahead.

Creating a Stored Program

To build a new stored PL/SQL program, you use one of SQL's *CREATE* statements. For example, if you want to create a stored function that counts words in a string, you can do so using a *CREATE FUNCTION* statement:

```
CREATE FUNCTION wordcount (str IN VARCHAR2)
  RETURN PLS_INTEGER
AS
  declare local variables here
BEGIN
  implement algorithm here
```

```
END;  
/
```

As with the simple BEGIN-END blocks shown earlier, running this statement from SQL*Plus requires a trailing slash on a line by itself.

Assuming that the DBA has granted you Oracle's CREATE PROCEDURE privilege (which also gives you the privilege of creating functions), this statement causes Oracle to compile and store this stored function in your schema. If your code compiles, you'll probably see a success message such as:

```
Function created.
```

If another database object, such as a table or package, named wordcount already exists in your Oracle schema, CREATE FUNCTION will fail with the error message *ORA-00955: name is already used by an existing object*. That is one reason that Oracle provides the OR REPLACE option, which you will want to use probably 99% of the time:

```
CREATE OR REPLACE FUNCTION wordcount (str IN VARCHAR2)  
    RETURN PLS_INTEGER  
AS same as before
```

The OR REPLACE option avoids the side effects of dropping and recreating the program; in other words, it preserves any object privileges you have granted to other users or roles. Fortunately, it replaces only objects of the same type, and it won't automatically drop a table named wordcount just because you decided to create a function by that name.

As with anonymous blocks used more than once, programmers generally store these statements in files in the operating system. I could create a file *wordcount.fun* for this function and use the SQL*Plus @ command to run it:

```
SQL> @wordcount.fun
```

```
Function created.
```

As mentioned earlier, SQL*Plus does not, by default, echo the contents of scripts. You can SET ECHO ON to see the source code scroll past on the screen, including the line numbers that the database assigns; this setting can be helpful when you're troubleshooting. Let's introduce an error into the program by commenting out a variable declaration (line 4):

```
/* File on web: wordcount.fun */  
SQL> SET ECHO ON  
SQL> @wordcount.fun  
SQL> CREATE OR REPLACE FUNCTION wordcount (str IN VARCHAR2)  
2     RETURN PLS_INTEGER  
3 AS  
4 /* words PLS_INTEGER := 0; ***Commented out for intentional error*** */  
5     len PLS_INTEGER := NVL(LENGTH(str),0);
```

```

6     inside_a_word BOOLEAN;
7 BEGIN
8     FOR i IN 1..len + 1
9     LOOP
10        IF ASCII(SUBSTR(str, i, 1)) < 33 OR i > len
11        THEN
12            IF inside_a_word
13            THEN
14                words := words + 1;
15                inside_a_word := FALSE;
16            END IF;
17        ELSE
18            inside_a_word := TRUE;
19        END IF;
20    END LOOP;
21    RETURN words;
22 END;
23 /

```

Warning: Function created with compilation errors.

This message tells us that the function was created, but that there were compilation errors that render it inoperable. We’ve succeeded in storing the source code in the database; now we need to tease the details of the error out of the database. The quickest way to see the full text of the error message is to use the SQL*Plus `SHOW ERRORS` command, abbreviated as `SHO ERR`:

```

SQL> SHO ERR
Errors for FUNCTION WORDCOUNT:
LINE/COL ERROR
-----
14/13   PLS-00201: identifier 'WORDS' must be declared
14/13   PL/SQL: Statement ignored
21/4    PL/SQL: Statement ignored
21/11   PLS-00201: identifier 'WORDS' must be declared

```

The compiler has detected both occurrences of the variable, reporting the exact line and column numbers. To see more detail about any server-based error, you can look it up by its identifier—PLS-00201 in this case—in Oracle’s *Database Error Messages* document.

Behind the scenes, `SHOW ERRORS` is really just querying Oracle’s `USER_ERRORS` view in the data dictionary. You can query that view yourself, but you generally don’t need to (see the following sidebar).

Show Other Errors

Many Oracle programmers know only one form of the SQL*Plus command:

```
SQL> SHOW ERRORS
```

and they incorrectly believe that they must query the `USER_ERRORS` view directly to see anything but the error messages from the most recent compile. However, you can append to `SHOW ERRORS` an object category and a name, and it will display the latest errors for any object:

```
SQL> SHOW ERRORS category [schema.]object
```

For example, to view the latest errors for the `wordcount` function, specify:

```
SQL> SHOW ERRORS FUNCTION wordcount
```

Use caution when interpreting the output:

```
No errors.
```

This message actually means one of three things: (1) the object did compile successfully; (2) you gave it the wrong category (for example, function instead of procedure); or (3) no object by that name exists.

The complete list of categories this command recognizes varies by version, but includes the following:

```
DIMENSION  
FUNCTION  
JAVA SOURCE  
JAVA CLASS  
PACKAGE  
PACKAGE BODY  
PROCEDURE  
TRIGGER  
TYPE  
TYPE BODY  
VIEW
```

It's common practice to append a `SHOW ERRORS` command after every scripted `CREATE` statement that builds a stored PL/SQL program. So, a “good practices” template for building stored programs in SQL*Plus might begin with this form:

```
CREATE OR REPLACE program-type  
AS  
    your code  
END;  
/  
  
SHOW ERRORS
```

(I don't usually include `SET ECHO ON` in scripts, but rather type it at the command line when needed.)

When your program contains an error that the compiler can detect, `CREATE` will still cause the Oracle database to store the program in the database, though in an invalid

state. If, however, you mistype part of the CREATE syntax, the database won't be able to figure out what you are trying to do and won't store the code in the database.

Executing a Stored Program

We've already looked at two different ways to invoke a stored program: wrap it in a simple PL/SQL block or use the SQL*Plus EXECUTE command. You can also use stored programs inside other stored programs. For example, you can invoke a function such as wordcount in any location where you could use an integer expression. Here is a short illustration of how I might test the wordcount function with a strange input (CHR(9) is an ASCII "tab" character):

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('There are ' || wordcount(CHR(9)) || ' words in a tab');
END;
/
```

I have embedded wordcount as part of an expression and supplied it as an argument to DBMS_OUTPUT.PUT_LINE. Here, PL/SQL automatically casts the integer to a string so it can concatenate it with two other literal expressions. The result is:

```
There are 0 words in a tab
```

You can also invoke many PL/SQL functions inside SQL statements. Here are several examples of how you can use the wordcount function:

- Apply the function in a select list to compute the number of words in a table column:

```
SELECT isbn, wordcount(description) FROM books;
```

- Use the ANSI-compliant CALL statement, binding the function output to a SQL*Plus variable, and display the result:

```
VARIABLE words NUMBER
CALL wordcount('some text') INTO :words;
PRINT :words
```

- Same as above, but execute the function from a remote database as defined in the database link test.newyork.ora.com.

```
CALL wordcount@test.newyork.ora.com('some text') INTO :words;
```

- Execute the function, owned by schema bob, while logged in to any schema that has appropriate authorization:

```
SELECT bob.wordcount(description) FROM books WHERE id = 10007;
```

Showing Stored Programs

Sooner or later you will want to get a list of the stored programs you own, and you may also need to view the most recent version of program source that Oracle has saved in

its data dictionary. This is one task that you will find far easier if you use some kind of GUI-based navigation assistant, but if you lack such a tool, it's not too hard to write a few SQL statements that will pull the desired information out of the data dictionary.

For example, to see a complete list of your programs (and tables, indexes, etc.), query the `USER_OBJECTS` view, as in:

```
SELECT * FROM USER_OBJECTS;
```

This view shows name, type, creation time, latest compile times, status (valid or invalid), and other useful information.

If all you need is the summary of a PL/SQL program's callable interface in SQL*Plus, the easiest command to use is `DESCRIBE`:

```
SQL> DESCRIBE wordcount
FUNCTION wordcount RETURNS BINARY_INTEGER
Argument Name                Type                In/Out Default?
-----
STR                           VARCHAR2            IN
```

`DESCRIBE` also works on tables, views, object types, procedures, and packages. To see the complete source code of your stored programs, query `USER_SOURCE` or `TRIGGER_SOURCE`. (Querying from these data dictionary views is discussed in further detail in [Chapter 20](#).)

Managing Grants and Synonyms for Stored Programs

When you first create a PL/SQL program, normally no one but you or the DBA can execute it. To give another user the authority to execute your program, issue a `GRANT` statement:

```
GRANT EXECUTE ON wordcount TO scott;
```

To remove the privilege, use `REVOKE`:

```
REVOKE EXECUTE ON wordcount FROM scott;
```

You could also grant the `EXECUTE` privilege to a role:

```
GRANT EXECUTE ON wordcount TO all_mis;
```

Or, if appropriate, you could allow any user on the current database to run the program:

```
GRANT EXECUTE ON wordcount TO PUBLIC;
```

If you grant a privilege to an individual like Scott, and to a role of which that user is a member (say, `all_mis`), and also grant it to `PUBLIC`, the database remembers all three grants until they are revoked. Any one of the grants is sufficient to permit the individual to run the program, so if you ever decide you don't want Scott to run it, you must revoke the privilege from Scott, and revoke it from `PUBLIC`, and finally revoke it from the `all_mis` role (or revoke that role from Scott).

To view a list of privileges you have granted to other users and roles, you can query the `USER_TAB_PRIVS_MADE` data dictionary view. Somewhat counterintuitively, PL/SQL program names appear in the `table_name` column:

```
SQL> SELECT table_name, grantee, privilege
       2 FROM USER_TAB_PRIVS_MADE
       3 WHERE table_name = 'WORDCOUNT';
```

TABLE_NAME	GRANTEE	PRIVILEGE
WORDCOUNT	PUBLIC	EXECUTE
WORDCOUNT	SCOTT	EXECUTE
WORDCOUNT	ALL_MIS	EXECUTE

When Scott does have the EXECUTE privilege on wordcount, he will probably want to create a synonym for the program to avoid having to prefix it with the name of the schema that owns it:

```
SQL> CONNECT scott/tiger
Connected.
SQL>CREATE OR REPLACE SYNONYM wordcount FOR bob.wordcount;
```

Now he can execute the program in his programs by referring only to the synonym:

```
IF wordcount(localvariable) > 100 THEN...
```

This is a good thing, because if the owner of the function changes, only the synonym (and not any stored program) needs modification.

It's possible to create a synonym for a procedure, function, package, or user-defined type. Synonyms for procedures, functions, or packages can hide not only the schema but also the actual database; you can create a synonym for remote programs as easily as local programs. However, synonyms can only hide schema and database identifiers; you cannot use a synonym in place of a packaged subprogram.

Removing a synonym is easy:

```
DROP SYNONYM wordcount;
```

Dropping a Stored Program

If you really, truly don't need a particular stored program anymore, you can drop it using SQL's DROP statement:

```
DROP FUNCTION wordcount;
```

You can drop a package, which can be composed of up to two elements (a specification and body), in its entirety:

```
DROP PACKAGE pkgname;
```

Or you can drop only the body without invalidating the corresponding specification:

```
DROP PACKAGE BODY pkgname;
```

Any time you drop a program that other programs call, the callers will be marked INVALID.

Hiding the Source Code of a Stored Program

When you create a PL/SQL program as previously described, the source code will be available in clear text in the data dictionary, and any DBA can view or even alter it. To protect trade secrets or to prevent tampering with your code, you might want some way to obfuscate your PL/SQL source code before delivering it.

Oracle provides a command-line utility called *wrap* that converts many CREATE statements into a combination of plain text and hex. It's not true encryption, but it does go a long way toward hiding your code. Here are a few extracts from a wrapped file:

```
FUNCTION wordcount wrapped
0
abcd
abcd ...snip...
1WORDS:
10:
1LEN:
1NVL:
1LENGTH:
1INSIDE_A_WORD:
1BOOLEAN: ...snip...
a5 b 81 b0 a3 a0 1c 81
b0 91 51 a0 7e 51 a0 b4
2e 63 37 :4 a0 51 a5 b a5
b 7e 51 b4 2e :2 a0 7e b4
2e 52 10 :3 a0 7e 51 b4 2e
d :2 a0 d b7 19 3c b7 :2 a0
d b7 :2 19 3c b7 a0 47 :2 a0
```

If you need true encryption—for example, to deliver information such as a password that really needs to be secure—you should not rely on this facility.⁹

To learn more about the wrap utility, see [Chapter 20](#).

Editing Environments for PL/SQL

As I mentioned earlier, you can use a “lowest common denominator” editing and execution environment like SQL*Plus, or you can use an integrated development environ-

9. Oracle does provide a way of incorporating true encryption into your own applications using the built-in package DBMS_CRYPTO (or DBMS_OBFUSCATION_TOOLKIT) in releases before Oracle Database 10g; see [Chapter 23](#) for information on DBMS_CRYPTO.

ment that offers extensive graphical interfaces to improve your productivity. This section lists some of the most popular IDE tools. I do not recommend any particular tool; you should carefully define your list of requirements and priorities for such a tool and then see which of them best meets your needs.

Product	Description
Toad	Offered by Quest Software, Toad is far and away the most popular PL/SQL IDE. Its free and commercial versions are used by hundreds of thousands of developers.
SQL Navigator	Also offered by Quest Software, SQL Navigator is used by tens of thousands of developers who love the product's interface and productivity features.
PL/SQL Developer	PL/SQL Developer, sold by Allround Automations , is a favorite of many PL/SQL developers. It is built around a plug-in architecture, so third parties can offer extensions to the base product.
SQL Developer	After years of little or no support for PL/SQL editing, Oracle Corporation created SQL Developer as a “fork” of the foundation JDeveloper tool. SQL Developer is free and increasingly robust.

There are many other PL/SQL IDEs out there, but those just listed are some of the best and most popular.

Calling PL/SQL from Other Languages

Sooner or later, you will probably want to call PL/SQL from C, Java, Perl, PHP, or any number of other places. This seems like a reasonable request, but if you've ever done cross-language work before, you may be all too familiar with some of the intricacies of mating up language-specific datatypes—especially composite datatypes like arrays, records, and objects—not to mention differing parameter semantics or vendor extensions to “standard” application programming interfaces (APIs) like Microsoft's Open Database Connectivity (ODBC).

I will show a few very brief examples of calling PL/SQL from the outside world. Let's say that I've written a PL/SQL function that accepts an ISBN expressed as a string and returns the corresponding book title:

```
/* File on web: booktitle.fun */
FUNCTION booktitle (isbn_in IN VARCHAR2)
  RETURN VARCHAR2
IS
  l_title books.title%TYPE;
  CURSOR icur IS SELECT title FROM books WHERE isbn = isbn_in;
BEGIN
  OPEN icur;
  FETCH icur INTO l_title;
  CLOSE icur;
  RETURN l_title;
END;
```

In SQL*Plus, I could call this in several different ways. The shortest way would be as follows:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(booktitle('0-596-00180-0'))
Learning Oracle PL/SQL
```

PL/SQL procedure successfully completed.

Next, I'll show you how I might call this function from the following environments:

- C, using Oracle's precompiler (Pro*C)
- Java, using JDBC
- Perl, using Perl DBI and DBD::Oracle
- PHP
- PL/SQL Server Pages

These examples are very contrived—for example, the username and password are hard-coded, and the programs simply display the output to stdout. Moreover, I'm not even going to pretend to describe every line of code. Still, these examples will give you an idea of some of the patterns you may encounter in different languages.

C: Using Oracle's Precompiler (Pro*C)

Oracle supplies at least two different C-language interfaces to Oracle: one called OCI (Oracle Call Interface), which is largely the domain of rocket scientists, and the other called Pro*C. OCI provides hundreds of functions from which you must code low-level operations such as open, parse, bind, define, execute, fetch... and that's just for a single query. Because the simplest OCI program that does anything interesting is about 200 lines long, I thought I'd show a Pro*C example instead. Pro*C is a precompiler technology that allows you to construct source files containing a mix of C, SQL, and PL/SQL. You run the following through Oracle's *proc* program, and out will come C code:

```
/* File on web: callbooktitle.pc */
#include <stdio.h>
#include <string.h>

EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR uid[20];
    VARCHAR pwd[20];
    VARCHAR isbn[15];
    VARCHAR btitle[400];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA.H;

int sqlerror();

int main()
{
    /* VARCHARs actually become a struct of a char array and a length */
```

```

strcpy((char *)uid.arr,"scott");
uid.len = (short) strlen((char *)uid.arr);
strcpy((char *)pwd.arr,"tiger");
pwd.len = (short) strlen((char *)pwd.arr);

/* this is a cross between an exception and a goto */
EXEC SQL WHENEVER SQLERROR DO sqlerror();

/* connect and then execute the function */
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
EXEC SQL EXECUTE
    BEGIN
        :btitle := booktitle('0-596-00180-0');
    END;
END-EXEC;

/* show me the money */
printf("%s\n", btitle.arr);

/* disconnect from ORACLE */
EXEC SQL COMMIT WORK RELEASE;
exit(0);
}

sqlerror()
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}

```

As you can see, Pro*C is not an approach for which language purists will be pining away. And trust me, you don't want to mess with the C code that this generates. Nevertheless, many companies find that Pro*C (or Pro*Cobol, or any of several other languages Oracle supports) serves as a reasonable middle ground between, say, Visual Basic (too slow and clunky) and OCI (too hard).

Oracle's own documentation offers the best source of information regarding Pro*C.

Java: Using JDBC

As with C, Oracle provides a number of different approaches to connecting to the database. The embedded SQL approach, known as SQLJ, is similar to Oracle's other pre-compiler technology, although a bit more debugger-friendly. A more popular and Java-centric approach is known as JDBC (which doesn't really stand for anything), although the usual interpretation is "Java Database Connectivity."

```

/* File on web: Book.java */
import java.sql.*;

```

```

public class Book
{
    public static void main(String[] args) throws SQLException
    {
        // initialize the driver and try to make a connection

        DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());
        Connection conn =
            DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:o92",
                                       "scott", "tiger");

        // prepareCall uses ANSI92 "call" syntax
        CallableStatement cstmt = conn.prepareCall("{? = call booktitle(?)}");

        // get those bind variables and parameters set up
        cstmt.registerOutParameter(1, Types.VARCHAR);
        cstmt.setString(2, "0-596-00180-0");

        // now we can do it, get it, close it, and print it
        cstmt.executeUpdate();
        String bookTitle = cstmt.getString(1);
        conn.close();
        System.out.println(bookTitle);
    }
}

```

This particular example uses the thin driver, which provides great compatibility and ease of installation (all the network protocol smarts exist in a Java library), at some expense of communications performance. An alternative approach would be to use what's known as the OCI driver. Don't worry: there's no rocket science programming required to use it, despite the name!

Perl: Using Perl DBI and DBD::Oracle

Much beloved by the system administration community, Perl is something of the mother of all open source languages. Now in version 5.10, it does just about everything and seems to run everywhere. And with nifty autoconfiguration tools such as CPAN (Comprehensive Perl Archive Network), it's a cinch to install community-supplied modules such as the DataBase Interface (DBI) and the corresponding Oracle driver, DBD::Oracle:

```

/* File on web: callbooktitle.pl */
#!/usr/bin/perl

use strict;
use DBI qw(:sql_types);

# either make the connection or die
my $dbh = DBI->connect(
    'dbi:Oracle:o92',
    'scott',

```

```

        'tiger',
        {
            RaiseError => 1,
            AutoCommit => 0
        }
    ) || die "Database connection not made: $DBI::errstr";

my $retval;

# make parse call to Oracle, get statement handle
eval {
    my $func = $dbh->prepare(q{
        BEGIN
            :retval := booktitle(isbn_in => :bind1);
        END;
    });

    # bind the parameters and execute
    $func->bind_param(":bind1", "0-596-00180-0");
    $func->bind_param_inout(":retval", \$retval, SQL_VARCHAR);
    $func->execute;

};

if( $@ ) {
    warn "Execution of stored procedure failed: $DBI::errstr\n";
    $dbh->rollback;
} else {
    print "Stored procedure returned: $retval\n";
}

# don't forget to disconnect
$dbh->disconnect;

```

Perl is one of those languages in which it is shamelessly easy to write code that is impossible to read. It's not a particularly fast or small language, either, but there are compiled versions that at least address the speed problem.

For more information about Perl and Oracle, see *Programming the Perl DBI* by Alligator Descartes and Tim Bunce. There are also many excellent books on the Perl language, not to mention the online information at perl.com (an O'Reilly site), perl.org, and cpan.org.

PHP: Using Oracle Extensions

If you are the kind of person who might use the free and wildly popular web server known as Apache, you might also enjoy using the free and wildly popular programming language known as PHP. Commonly employed to build dynamic web pages, PHP can also be used to build GUI applications or to run command-line programs. As you might expect, Oracle is one of many database environments that work with PHP; Oracle Cor-

poration has, in fact, partnered with Zend in order to provide a “blessed” distribution of the Oracle database with PHP.¹⁰

This example uses the family of PHP functions known as OCI8. Don’t let the “8” in the name fool you—it should work with everything from Oracle7 to Oracle Database 11g:

```
/* File on web: callbooktitle.php */
<?PHP
// Initiate the connection to the o92 database
$conn = OCILogon ("scott", "tiger", "o92");

// Make parse call to Oracle, get statement identity
$stmt = OCIParse($conn,
    "begin :res := booktitle('0-596-00180-0'); end;");

// Show any errors
if (!$stmt) {
    $err = OCIError();
    echo "Oops, you broke it: ".$err["message"];
    exit;
}

// Bind 200 characters of the variable $result to placeholder :res
OCIBindByName($stmt, "res", &$result, 200);

// Execute
OCIExecute($stmt);

// Stuff the value into the variable
OCIResult($stmt,$result);

// Display on stdout
echo "$result\n";

// Relax
OCILogoff($conn);
?>
```

When executed at the command line, it looks something like this:

```
$ php callbooktitle.php
Learning Oracle PL/SQL
```

By the way, these Oracle OCI functions are not available in PHP by default, but it shouldn’t be too difficult for your system administrator to rebuild PHP with the Oracle extensions.

10. Note that if you want support for PHP, you will need to get it from the user community or from a firm like Zend. Oracle Corporation does not take support calls for PHP.

You can find more information about PHP at php.net or in one of O'Reilly's many books on the subject. For PHP tips specific to Oracle, visit the [Oracle Technology Network](#).

PL/SQL Server Pages

Although the PL/SQL Server Pages (PSP) environment is proprietary to Oracle, I thought I would mention it because it's a quick way to get a web page up and running. PSP is another precompiler technology; it lets you embed PL/SQL into HTML pages. The `<%= %>` construct here means "process this as PL/SQL and return the result to the page."

```
/* File on web: favorite_plsql_book.psp */
<%@ page language="PL/SQL" %>
<%@ plsql procedure="favorite_plsql_book" %>
<HTML>
  <HEAD>
    <TITLE>My favorite book about PL/SQL</TITLE>
  </HEAD>
  <BODY>
    <%= booktitle( '0-596-00180-0' ) %>
  </BODY>
</HTML>
```

When properly installed on a web server connected to an Oracle database, this page displays as in [Figure 2-3](#).

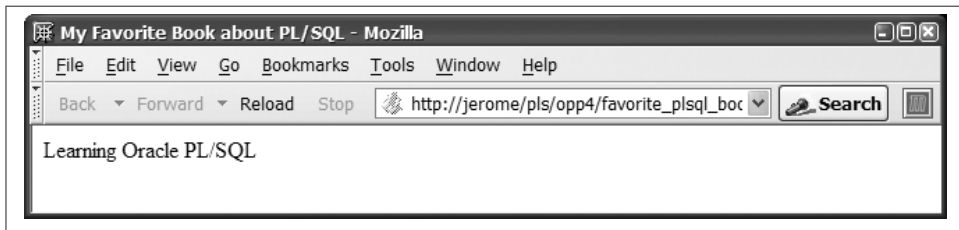


Figure 2-3. Output from a PL/SQL Server Page

I'm rather fond of PL/SQL Server Pages as a good way to put together data-driven websites fairly quickly.

For more information about PL/SQL Server Pages, see [Learning Oracle PL/SQL](#), which is by the authors of the book you're reading now.

And Where Else?

You've seen how to use PL/SQL in SQL*Plus and in a number of other common environments and programming languages. There are still more places and ways that you can use PL/SQL:

- Embedded in COBOL or FORTRAN and processed with Oracle's precompiler
- Called from Visual Basic, using some flavor of ODBC
- Called from the Ada programming language, via a technology called SQL*Module
- Executed automatically, as triggers on events in the Oracle database such as table updates
- Scheduled to execute on a recurring basis inside the Oracle database, via the DBMS_SCHEDULER supplied package
- In the TimesTen database, an in-memory database acquired by Oracle Corporation, whose contents can be manipulated with PL/SQL code, just like the relational database

I am not able, (un)fortunately, to address all these topics in this book.