CHAPTER 28

Creating Stored Procedures with Oracle8i

In this chapter

Introducing PL/SQL

Writing stored procedures in PL/SQL

Creating a package

In this chapter, I'm going to discuss the details of how to create various procedural objects, such as stored procedures and functions. I'll also discuss Oracle's implementation of SQL and the extensions necessary to provide a rich programming environment.

Introducing PL/SQL

Every database vendor has its own variation of the SQL language, and Oracle is no exception. Procedural Language/SQL, also known as PL/SQL, contains a series of extensions that make it possible to build efficient programs which can perform complex database functions.

PL/SQL is used primarily to build stored procedures and functions. These routines essentially become an extension to the PL/SQL language. Thus, you can call your own stored procedure as easily as you would use a **Select** statement. In fact, it may be easier to call a stored procedure than use a **Select** statement, since stored procedures typically have fewer parameters.

Comments

Comments in PL/SQL are the same as SQL Server's T-SQL. Double hyphen comments (`--`) mark the start of a comment whose text continues to the end of the line. Slash asterisk, asterisk slash comments (`/* */`) can be used anywhere a space can be used, and may span multiple lines.

Constants

String values must be enclosed in single quotes ('), as shown below:

```
'This is a string constant.'
```

If you wish to embed a single quote in a string constant, simply use two single quotes together, like this:

```
'This is a ''string'' with embedded single quotes.'
```

This string constant would be understood by PL/SQL as:

```
This is a 'string' with embedded single quotes.
```

Quote'th the raven, never double: Double quotes (") are not the same thing as two single quotes (' '). Double quotes are used to indicate case-sensitive identifiers, while two single quotes inside a string constant indicate that a single single quote should be inserted at that position.

Numbers, as you might expect, may begin with a plus sign (+) or a minus sign (-), followed by a series of numeric digits, a decimal point, and more digits. Some examples of numbers are:

```
0          3.1415926    1000.0       -100.001     +512
```

Dates, which are stored internally in a special format, are written like strings, except theyare formatted as DD-MM-YY. For example,

```
'27-Jul-65'  '27-Jul-1965'   '1-May-2000'
```

are valid **Date** constants.

Identifiers

An identifier can be up to 30 characters in length. They must begin with a letter and may contain letters, numbers, and the underscore character (_). Identifiers are used as the name of a stored procedure or function, as a local variable, and as the name of various database objects. Double quotes (") surround identifiers that contain spaces or special characters.

Variables

PL/SQL allows you to define local variables that can be used in stored procedures and functions. They are declared in the **Declare** section of your procedure. You can

declare a variable to be of any data type supported by Oracle8*i*. Unlike SQL Server, variables need not begin with a special character. Any identifier that doesn't conflict with an Oracle8*i* keyword may be used.

See the discussion of Oracle8*i* data types in Chapter 26 if you would like to become more familiar with them.

Functions

PL/SQL provides a number of functions that can be used to perform calculations or data conversions. Some of the more interesting functions are listed in Table 28-1.

Table 28-1
Selected Functions in PL/SQL

Function	*Description*
Add_months	Adds the specified number of months to the specified date.
ASCII	Returns the ASCII code of the left-most character of the specified character string.
Ceil	Returns the smallest integer value greater than or equal to the specified value.
Chr	Returns the character corresponding to the specified numeric ASCII code value.
Floor	Returns the largest integer value less than or equal to the specified value.
Initcap	Capitalizes the first character of each word in a string.
Instr	Returns the position of the specified search string in a specified data string.
Last_day	Returns the date of the last day of the month for the specified data value.
Length	Returns the size of a string, number, date, or expression.
Lower	Converts all uppercase characters to lowercase.
Ltrim	Removes leading blanks from a character string.
Mod	Computes the remainder after dividing the two values.

Continued

Table 28-1 *(continued)*

Function	*Description*
Months_between	Computes the number of months between two dates.
New_time	Converts a **Date** value to the specified time zone.
Next_day	Returns a **Date** value containing the day that follows the specified date.
Replace	Replaces the search string with a replacement string in the specified string.
Round	Rounds the specified **Number** or **Date** value to the specified accuracy.
Rtrim	Removes trailing blanks from a character string.
Substr	Returns a string of characters from the specified string with the specified starting location and length.
Sysdate	Returns the current date and time.
To_char	Converts the specified value to a character string.
To_date	Converts the specified value to a **Date** value.
To_number	Converts the specified value to a numeric value.
Trunc	Truncates the specified **Date** or **Number** value using the specified accuracy.
Upper	Converts all lowercase characters to uppercase.
User	Contains the user name of the current user.

If you don't like these, then build your own: If you need a function that isn't available in PL/SQL, you can easily build your own using the **Create Function** statement. Functions are a variation on stored procedures. The only difference is that a function returns a single value that can be used as part of an expression, while a stored procedure can't be used as part of an expression.

Block structure

The *block structure* is the fundamental way statements are organized into a stored procedure or function. A block structure is broken into three main sections, the **Declare** section, the main body and the **Exception** section. The **Declare** section is

used to declare block-wide variables. These variables can be used in between the **Begin** and **End** statements. You may optionally assign an initial value for these variables.

The **Begin** statement marks the start of the executable commands section of the block. When the block is called, control will begin with the first statement following the **Begin** statement. Program flow will continue until it reaches either the **End** statement or the **Exception** statement. When either of these statements is reached, execution is complete and control will return to the calling program.

If an error occurs while running in the executable commands section of the block, control will be transferred to the first statement following the **Exception** statement. If the **Exception** section isn't present, an error message will be returned to the calling program. Once you transfer control the to **Exception** section, you can't return to the executable commands section.

Nesting: A block can be used anywhere a PL/SQL statement can be used. Thus, you can nest one block inside of another. In the innermost block, you can use any of the variables declared in the outer blocks. However, you can't use any of the variables declared in an inner block once you are outside that block. The syntax for a block is:

```
<< <block_name> >>
[Declare
   <variable> <datatype> [:= <initial_value>];
   [<variable> <datatype> [:= <initial_value>];] ...
]

Begin
   <statement>;
   [<statement>;]...
[Exception
   When <condition> Then <statement>; [<statement>;]...
   [When <condition> Then <statement>; [<statement>;]...]...
]
```

End [<block_name>]; where <block_name> is an identifier that is associated with the block; <variable> is an identifier that will be used to store information locally in the block; .<datatype> is any legal Oracle8*i* data type; <initial_value> is a constant that is appropriate for the data type; <statement> is any legal PL/SQL statement or command; and <condition> is an exception (see Table 28-2), a list of exceptions that are **Or**'ed together, or the keyword **Others**, which traps any remaining exceptions.

Table 28-2
Exceptions

Exception	*Description*
CURSOR_ALREADY_OPEN	An **Open** statement tried to open a cursor that was already open.
DUP_VAL_ON_INDEX	An **Insert** or **Update** statement created a duplicate value in a **Unique** index.
INVALID_CURSOR	An **Open** statement tried to open an undefined cursor; a **Close** statement tried to close a closed cursor; a **Fetch** statement tried to use an unopened cursor, and so on.
INVALID_NUMBER	An illegal numeric value was found when trying to convert a character string to a numeric value.
LOGIN_DENIED	The user name and password combination was invalid in a **Connect** statement.
NO_DATA_FOUND	A **Select** statement returned zero rows.
NOT_LOGGED_ON	An attempt was made to access database resources without being connected to the database.
PROGRAM_ERROR	A catchall error used by PL/SQL to trap its own errors.
STORAGE_ERROR	Insufficient memory was available to execute the function, or the available memory was corrupted (possible subscripting error).
TIMEOUT_ON_RESOURCE	A resource wasn't available when it should have been.
TOO_MANY_ROWS	A **Select** statement that should return a single row returned more than one row.
TRANSACTION_BACKED_OUT	A remote part of a transaction failed and was rolled back.
VALUE_ERROR	A conversion error, a truncation error, or a precision error affecting a variable or column value occurred
ZERO_DIVIDE	An attempt was made to divide by zero.

Blockhead: The name of a block is an optional feature that marks the beginning of a block. It is simply an identifier that is enclosed in double less than (<<) and double greater than (>>) signs, such as <<MyBlock>>. The same name that begins the block must also be specified in the **End** statement, like this:`End MyBlock`.

Procedures, functions, and packages

Procedures and functions contain code that can be treated as an extension to PL/SQL. A procedure can be used in much the same way as a command or SQL statement, while a function can be incorporated into any expression. Procedures and functions can be written as standalone routines or combined in a single unit called a *package*. (The **Create Package** statement is explained at the end of this chapter.)

Just a routine check: Procedures and functions are basically the same thing. The only difference is in how they are used. Functions can be used within an expression, while a procedure must be called as a separate statement I use the term "routine" to refer to something that can be either a procedure or a function. For instance, I might say that "Routines have parameters" rather than saying "Procedures and functions have parameters". Not only does this make things a little clearer, it also saves me a lot of typing.

Procedures and functions

Procedures and functions are just blocks with a header that defines the name of a particular routine and the list of parameters associated with it. A procedure is similar to a Visual Basic subroutine in how it is used. You pass a series of parameters to a procedure as a single statement. When it finishes, control is returned to the next statement in your program.

A PL/SQL function works like a Visual Basic function. It is used in an expression to compute a value based on a set of parameters. When the function returns its value, the rest of the expression is processed.

The syntax for a procedure definition is:

```
Procedure [<username>.]<procedure>
   [(<argument> [,<argument>]...)]
   {Is|As} <block>
```

The syntax for a function definition is:

```
Function [<username>.]<function>
   [(<argument> [,<argument>]...)]
   Return <datatype>
   {Is|As} <block>
```

where `<username>` is the name of the user associated with the procedure or function; `<procedure>` is the name of the procedure; `<function>` is the name of the function;`<argument>` is `<parameter> [In|Out|In Out] <datatype>`, where `<parameter>` is the name of the parameter; **In** means that the parameter is passed to the routine, but any changes in the parameter are not returned to the calling

program, **Out** means that no value is passed to the routine, but the routine will return a value to the calling program, **In Out** means that a value is passed to the routine and any changes in the parameter will be returned to the calling program, and <datatype> is the data type associated with the parameter; and

<block> is a block declaration as I discussed in the Block structure section earlier in this chapter. It is separated from the rest of the statement by using the **Is** or **As** keywords.

Return statement

The **Return** statement is used to return a value to the calling program. It uses the following syntax:

```
Return(<value>);
```

where <value> is a variable or expression containing the information that will be returned as the value of the function.

Packages

A *package* is merely a single unit containing a collection of one or more procedures and functions (or routines), with some optional global variables. It corresponds to a Visual Basic module. The package consists of three parts: global declarations, which are optional, at least one routine, and a block of code that is executed each time a routine in the package is called. This block is executed first, which allows you to initialize global variables, open a cursor, or any other logic that is common to all of the routines.

I love packages: Besides the obvious benefit of creating a single installation unit that combines many different routines, packages are often more efficient than independent stored procedures and functions, since the code is compiled together as a single unit. Thus, you avoid the extra costs of locating the new routine, loading it, and preparing to run it. All of this work is done when the first routine in the package is called.

Expressions

You can compute a single value based on a collection of local variables, parameters, functions, and columns retrieved from a table, and then assign the value to a local variable or column. PL/SQL uses the assignment operator (:=) to perform the assignment, as shown below:

```
MyVariable := 'A string value';
MyNumber := 3.14159265;
MyNumber := MyNumber * 20;
```

Flow control

Oracle8*i* /SQL supports a wider range of flow control statements than Microsoft's SQL Server. You can use statements such as **If**, **For,** and **While** to control the flow through your stored procedure. You can also call procedures and functions directly without having to use a special statement.

If statement

The **If** statement has the following syntax:

```
If <boolean_expression>
   {<statement>|<block>}
[Elsif <Boolean_expression>
   {<statement>|<block>}]...
[Else
   {<statement>|<block }]
End If;
```

<boolean_expression> is a boolean expression that is either True or False. If the Boolean expression is **True**, then the statement or block that immediately follows the expression will be executed. Otherwise, the statement or block that immediately follows the **Else** clause will be executed.

<statement> is an SQL statement or PL/SQL command.

<block> is a PL/SQL block. While variables and exceptions can be included, typically all you would use is the **Begin** and **End** statements to enclose a group of statements to be executed if <boolean_expression> is **True**.

Exit statement

The **Exit** statement allows you to exit from a loop or from a block. Its syntax is:

```
Exit [<block_name>] [When <boolean_expression>];
```

<block_name> is the name of a block (such as, <<myblock>>). If <block_name> isn't specified, the processing will resume with the statement that immediately follows the next **End** statement. Otherwise, processing will resume with the **End** statement that matches the specified <block_name>.

<boolean_expression> is an expression that, when **True**, will exit a loop or block of code, Otherwise processing will continue normally.

Loop statement

The **Loop** statement has the following syntax:

```
Loop
   <statement>
   [<statement>]
End Loop;
```

where <statement> is a valid PL/SQL statement.

Note that this looping statement creates an infinite loop. The only way to leave the loop is to use an **Exit** statement.

While statement

While statements are constructed with the following syntax:

```
While <boolean_expression>
   Loop
   <statement>
   [<statement>]
End Loop;
```

<boolean_expression> is evaluated at the start of each loop. As long as this expression is **True**, the statements contained in the loop will be executed. If the expression is **False**, execution will resume with the statement after the **End Loop** statement.

<statement> is a valid PL/SQL statement.

For statement

The **For** statement should be familiar if you're a Visual Basic programmer. However, its syntax is a little different than that which Microsoft uses:

```
For <variable> In <start_expression> .. <end_expression>
   Loop
   <statement>
   [<statement>]...
End Loop;
```

where <variable> is the variable to be incremented; <start_expression> is the initial value for the **For** variable; and<end_expression> is the ending value for the **For** variable.

Unlike the Visual Basic **For** statement, the PL/SQL **For** statement allows you to increment only the **For** variable.

For cursors only: See the section on Cursors later in this chapter to see how to increment a cursor.

Cursors

Cursors are a special type of variable that allow you to access the contents of a **Select** statement one row at a time. You can move the cursor through the result set and perform various operations on the columns retrieved, including updating values, deleting rows, and inserting rows.

The Cursor statement

The **Cursor** statement defines a cursor. Its syntax is:

```
Cursor <cursor>
   [Is <select_statement>];
```

where `<cursor>` is the name of the cursor variable; and`<select_statement>` is a **Select** statement that returns rows to be accessed through the cursor.

Defining a cursor merely creates the data structures necessary to access information from the database. No rows are actually retrieved until the **Open** statement is executed. However, information about the columns retrieved is available and can be accessed by using the **%Rowtype** and **%Type** cursor attributes.

Cursor attributes

You can determine additional information about a cursor by using a cursor attribute. Cursor attributes are appended to a cursor variable, such as:

```
MyCursor%Found
```

which will return a value indicating whether the last operation that used it was successful. You can use the combination of cursor and attribute anywhere you can use a variable.

Table 28-3 lists the attributes available for each cursor. Note that you need not do anything to make these attributes available. They are automatically present once a cursor has been defined.

Table 28-3
Cursor Attributes

Attribute	*Description*
%Found	Is **True** when the last operation (**Select**, **Insert**, **Update**, or **Delete**) was successful.
%Isopen	Is **True** when the cursor is open.
%Notfound	Opposite of **%Found**.
%Rowtype	Returns a record variable containing the same structure as the entire row in the table.
%Type	Returns the data type of the selected column.

The **%Rowtype** and **%Type** attributes are used in the **Declare** section as a data type for other variables. This allows you to declare a variable for a column or for the entire row without necessarily knowing their data type. You reference a particular column by using `<rowtype_variable>.<column_name>`. Consider the following code fragment:

```
Declare
   Cursor MyCursor Is
      Select MyColumn, AnotherColumn From MyTable;
   MyRow MyCursor%Rowtype;

Begin
Open MyCursor;

If MyRow.MyColumn = 0
   /* insert processing statements here */

Else
   /* insert processing statements here */

End;

Close MyCursor;
End;
```

`MyRow.MyColumn` is used to retrieve information from the MyColumn column from the MyTable table in the database.

The **%Found** and **%Notfound** attributes are extremely useful when managing loops. You can retrieve rows of information from the database in a **While** loop using the **%Notfound** attribute, as in the following code fragment:

```
Open MyCursor
While MyCursor%Notfound
   Loop
   Fetch MyCursor Into MyVariable;

   /* insert processing statements here */

   End Loop;
```

As long as there are rows remaining to be fetched, this loop will process each row. If the cursor didn't return any rows, the processing loop would be skipped.

Open statement

Before you access a cursor, you must use the **Open** statement. The syntax follows:

```
Open <cursor>;
```

where `<cursor>` is the name of a cursor that has already been declared.

When the **Open** statement is executed, the **Select** statement associated with the cursor is executed and the information is made available. You can then use the **Fetch** statement to retrieve the information, and the **Delete** or **Update** statements to modify the information. (The **Fetch** statement is covered later in this chapter.)

But I did open it: Trying to access a cursor that hasn't been opened will generate an error. Also, trying to open a cursor that is already open will generate an error. You can use the **%IsOpen** attribute to verify that your cursor is in the proper state before trying to use it. This is extremely important if you are implementing code in the Exceptions section and you may not know the exact state of the cursor.

Close statement

The **Close** statement deallocates the resources associated with the cursor. Its syntax is simple:

```
Close <cursor>;
```

`<cursor>` is the name of a cursor that has already been declared.

Once a cursor has been closed, you need to open it again before you can use it.

Fetch statement

The **Fetch** statement is used to retrieve the next record into a variable for local access. It has the following syntax:

```
Fetch <cursor> Into {<record>|<variable> [,<variable>]...};
```

where

<cursor> is an open cursor; <record> is a variable declared using <cursor> **%Rowtype** as its data type; and<variable> is a variable whose data type is compatible with the data type of the corresponding column from the cursor. The list of variables must match the list of columns retrieved from the cursor in both number and data types.

For statement

Another variation of the **For** statement you saw earlier in "Flow Control" makes it easy to process all of the rows retrieved from the database. You should use the following syntax:

```
For <record> In <cursor>
```

where <record> is a variable declared using <cursor>**%Rowtype**,and <cursor> is an open cursor.

Consider the following code fragment:

```
Open MyCursor;
For MyRow In MyCursor
   Loop

   /* insert processing statements here */

   End Loop;
```

The statements in the **Loop** body will be processed for each row retrieved from the database. You don't need a **Fetch** statement to retrieve the row into a variable. The **For** statement handles that for you automatically. Using the record variable also simplifies your code, since you don't have to worry about ensuring that each individual column is specified correctly in the **Fetch** statement.

Update statement

You should use the following syntax when updating information retrieved by using a cursor:

```
Update <table>
   Set <column> = <value> [, <column>=<value>] ...
   Where Current Of <cursor>
```

where <table> is the name of the table you want to update; <column> is the name of the column you want to update; <value> is the value you want to assign to the column; and <cursor> is an open cursor containing the information you want to updated.

The **Update** statement affects only the information in the current row. No other rows are affected when the **Where Current Of** cursor clause is included. Note that in order to perform the update, you need to ensure that your cursor accesses an updateable view. This means that you may only access one table at a time when you declare the cursor.

Delete statement

Using the **Delete** statement, you can delete the current row pointed to by the cursor, with the following syntax:

```
Delete From <table>
   Where Current Of <cursor>
```

where `<table>` is the name of the table containing the row you want to delete, and `<cursor>` is an open cursor containing the information you want to delete.

Like the **Update** statement, the **Delete** statement may only reference a single table and the **Where Current Of** clause ensures that only the currently fetched row is affected by the statement.

Transactions

By now, you understand the importance of using transactions in your application where you need to ensure multiple changes are performed together as a single atomic unit. So it should come as no surprise that PL/SQL also includes support for transaction. As you would expect, there is a statement to mark the beginning of the transaction and another statement to mark the end of the transaction.

Set Transaction statement

The **Set Transaction** statement marks the beginning of a transaction and has the following syntax:

```
Set Transaction {Read Only|Use Rollback Segment <segment>}
```

where `<segment>` is the name of a rollback segment that is used by the transaction to hold undo information.

The **Set Transaction** statement must be the first statement in your transaction. You can ensure this by executing the **Commit** statement before executing the **Set Transaction** statement. (The **Commit** statement is covered later in this chapter.) The **Read Only** clause is used to ensure that records you read will always be consistent, though you will be prohibited from updating any of this data. If you want to update this data, you must specify the **Use Rollback Segment** clause and specify the rollback segment associated with your application. Each **Set Transaction** statement must be matched with the appropriate **Rollback** or **Commit** statement. (The **Rollback** statement is covered later in this chapter.)

Savepoint statement

The **Savepoint** statement allows you to mark a place in your transaction where you may choose to rollback your work. Its syntax follows:

```
Savepoint <savepoint>
```

where<savepoint> is an identifier that uniquely identifies the savepoint location.

Commit statement

The **Commit** statement saves all of the database changes made to the database since the **Set Transaction** statement was executed. It has the following syntax:

```
Commit [Work];
```

The **Work** keyword is optional and has no real meaning. It exists solely to comply with the ANSI SQL standard.

Rollback statement

The **Rollback** statement discards all of the changes made by a transaction to the database. This statement has the following syntax:

```
Rollback [Work] [To [Savepoint] <savepoint>]
```

If you rollback to the specified <savepoint>, all changes done after the **Savepoint** statement are discarded. All work done prior to the **Savepoint** remains uncommitted. You must use a **Commit** statement to save the changes or a **Rollback** statement without specifying a savepoint.

Other useful statements

Besides the statements and commands discussed so far, there are a few others that you may find useful.

DBMS_Output Package

PL/SQL includes a package to assist you with sending output to the console. This package is mostly useful when debugging your stored procedures using SQL*Plus. The package consists of three stored procedures with the following format:

```
DBMS_OUTPUT.PUT (<value>);
DBMS_OUTPUT.PUT_LINE (<value>);
DBMS_OUTPUT.NEW_LINE;
```

where <value> is a value to be printed on the console.

The **Put** routine displays a single value on the console. The output cursor remains on the same line, so that another call to **Put** will display another value next to the first. The **New_Line** routine advances the cursor to the first position in the next line. **Put_Line** is the equivalent of calling **Put**, immediately followed by **New_Line**.

These routines are controlled by the SERVEROUTPUT feature that you manage with the **Set** statement. Turning **On** this feature means that the output will be sent to the console, while **Off** means that the output will be discarded.

```
SET SERVEROUTPUT {ON|OFF}
```

Setting the **Serveroutput** feature outside your routines means that you can leave the debugging code in the routines. If you feel that you need to trace the routines execution, you can **Set Serveroutput On** and call the routine in SQL*Plus. If you don't want to view your debugging code, simply **Set Serveroutput Off**.

Raise statement

Handling errors is always interesting, especially when you have stored procedures calling other stored procedures. Sometimes you find an error condition where you want to kill the entire transaction or stored procedure. The **Raise** statement allows you to trigger an error condition.

The syntax for the **Raise** statement is:

```
Raise [<exception>]
```

where `<exception>` is an exception value selected from Table 28-2. If a value for `<exception>` is specified, the code in the current block's Exception section will be triggered. You can omit `<exception>` only when you are already processing an error in the Exception section and wish to pass the error onto the Exception section of the block that encloses the current block.

Creating Stored Procedures

Creating a stored procedure, function, or package in Oracle8*i* involves building a **Create Procedure**, **Create Function,** or **Create Package** statement and running it. While you can do this with DBA Studio, you can also use any of the SQL*Plus variations.

Creating a procedure or function

The **Create Procedure** statement is used to create a stored procedure, while the **Create Function** statement is used to create a stored function. The syntax for these are:

```
Create [Or Replace]
{<procedure_definition>|<function_definition>}
```

where <procedure_definition> is the syntax for a procedure as described above, beginning with **Procedure** and ending with **End**; and <function_definition> is the syntax for a function as described above, beginning with **Function** and ending with **End**;.

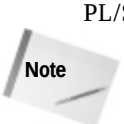
If you specify the **Or Replace** clause, the routine will be replaced with the new routine in <procedure_definition> or <function_definition>.

Creating a package

The **Create Package** statement is used to create a package. Its syntax is:

```
Create [Or Replace] Package Body <package> As
   [<variable> <datatype> [:= <initial_value>];] ...
   {<procedure_definition>|<function_definition>}
   [{<procedure_definition>|<function_definition>}]...
   [Begin
      <statement>
      [<statement>]...
      ]
   End [<package>];
```

where <variable> is an identifier that will be used to store information locally in the block; <datatype> is any legal Oracle8*i* data type; <initial_value> is a constant that is appropriate for the data type; <procedure_definition> is the syntax for a procedure as described above, beginning with **Procedure** and ending with **End**; <function_definition> is the syntax for a function as described above, beginning with **Function** and ending with **End**; and <statement> is any legal PL/SQL statement or command.



A happy ending: The last **End** statement in your routine should include the routine's name as the <block_name>, such as `End MyProcedure;` . This will prove extremely useful when trying to identify the beginning and end of a routine in the package.

Thoughts on Oracle8*i* Stored Procedures

I prefer to use three tools to create and test my stored procedures. I use a tool like Write to actually code the SQL statements. Then I use SQL*Plus for DOS to load the statements from the file and add them to my database. I leave the Write session active, so that I can correct syntax errors or add new functions while I have the SQL*Plus session. That way, all I have to do is save the file and reload it in SQL*Plus.

Once the stored procedure (or function or package) is loaded into my database, I execute the stored procedure directly in SQL*Plus. This lets me review the results interactively. I also like to use the **DBMS_OUTPUT** package to sprinkle my code with debugging statements. After all, executing the **Set Serveroutput** command allows me to quickly turn the information off or on, depending on how bad my luck is running.

After I'm satisfied that the procedure it doing what is should be doing, I create a simple Visual Basic program to verify it. Just because my procedure works with Oracle's tools doesn't mean that it will automatically work with Microsoft's tools. Once the program works, I'll move the procedure to my application and test it over again.

Now if you believe that I always follow this process, I've got a great deal for you on a bridge in San Francisco. But I do keep the tools lying around in case of problems, which occur more frequently than you might expect.

Summary

In this chapter you learned:

- about the language elements in PL/SQL language.
- about the key statements of the PL/SQL language.
- how to create transactions in PL/SQL.
- how to create stored procedures and functions in PL/SQL.
- how to create packages in PL/SQL.

✦ ✦ ✦