PL/SQL Tutorial

Learn the basics of PL/SQL

Ben Brumm

www.databasestar.com

Are you interested in learning PL/SQL?

You've come to the right place.

PL/SQL is not a hard language to learn, and with this tutorial, you'll be up and running quickly with writing your first PL/SQL program and moving on to more features of this language.

So, why should you read this tutorial when there's a lot of other tutorials, books, and videos out there?

- It's easy to follow.
- You'll start writing your first PL/SQL program very early, before reading a ton of pages on theory.
- It includes quiz questions and answers.
- It includes code samples which you can run yourself.
- It looks great.
- There are no ads.

Table of Contents

What's included in this tutorial?

This tutorial is broken down into several chapters, each of which cover a different area of PL/SQL. You'll start with the basics and move on to more features of the language.

Procedures and functions are common objects that use PL/SQL code. You'll learn what they are and how to create them in this chapter. You'll also learn what exceptions are, how to create them, and how to handle them.

Chapter 5: Cursors, arrays, inserting and updating data in PL/SQL.

PL/SQL allows you to insert or update data using SQL, and you'll see examples of how to do that in this chapter. You'll also learn what cursors are, how to use them, and what arrays are.

Chapter 6: Record types, bulk collect, and collections.

Using record types in PL/SQL can improve your code and you'll learn what they are and how to use them in this chapter. You'll learn what collections are, the different types of collections, and what the BULK COLLECT feature is.

Chapter 7: Packages and nested blocks

The final chapter will explain what nested blocks are and how to create them. You'll also learn what packages are, the difference between the specification and the body, why to use them, and how to create them.

Further Resources

If you've finished this tutorial and want more places to learn PL/SQL, check these out:

- Oracle Dev Gym: this website contains many quizzes for PL/SQL code which can help improve your knowledge and let you practice PL/SQL.
- Steve Feuerstein's PL/SQL blog: Steve's blog contains a range of great posts and tips on PL/SQL.
- StackOverflow questions: Read some of the older questions or try to answer some of the newer questions on PL/SQL.

Let's get started with Chapter 1!

Chapter 1: Introduction and Your First PL/SQL Program

In this chapter, you'll learn:

- What PL/SQL is and what it stands for
- The basic structure and syntax of a PL/SQL program
- Write and run your first PL/SQL program

Let's get right into it!

What Is PL/SQL?

PL/SQL stands for Procedural Language/Structured Query Language, and is an expansion of the SQL language developed by Oracle. It includes a set of procedural features (IF THEN ELSE logic, loops, variables) in addition to the SQL language. The code is written and executed on an Oracle database.

The SQL language includes many commands and features for reading and updating data in your database, such as SELECT, INSERT, UPDATE, and DELETE. Many applications and websites are built using SQL to interact with a database. These applications can do all sorts of things such as:

- Updating user profiles with an UPDATE statement
- Displaying forum posts with a SELECT statement
- Showing bank transactions with a SELECT statement
- Adding new products with an INSERT statement

What about all the business logic that is needed along with this?

Business logic is needed for things such as:

- Checking for existing users before adding a new user
- Validating account numbers are valid
- Recording deleted records in a backup or audit table rather than just deleting them

This logic is often added into the application code: in PHP, C#, ASP, or many other languages.

But you can also use PL/SQL for this logic. You'll learn how to do that in this tutorial.

Why Use PL/SQL?

You don't need to use PL/SQL to add business logic to your applications, but there are several reasons you may want to.

Implementation Details are Stored with the Database

When you write your business logic and implementation details on the database, it's closely tied to the database that uses it. Bryn Llewellyn writes here:

"The implementation details are the tables and the SQL statements that manipulate them. These are hidden behind a PL/SQL interface. This is the Thick Database paradigm: select, insert, update, delete, merge, commit, and rollback are issued only from database PL/SQL. Developers and end-users of applications built this way are happy with their correctness, maintainability, security, and performance."

Improved Performance

Storing your business logic in PL/SQL code means that it may perform better than application code because PL/SQL is closely tied to SQL and the Oracle database. It all runs on the same server, which in theory will provide a performance improvement.

Consistent Across Many Front-Ends

If you have your business logic stored on the database, you can provide that to applications that use it. You can have different applications use this database, and if they all use the same PL/SQL function then they will all manipulate data in the same way.

There are several disadvantages to using PL/SQL, such as being harder to manage source control, splitting logic between applications and databases making it harder to manage, and being tied to an Oracle database. However I think these are minor disadvantages, as source control with PL/SQL code is fairly good, and it's not very often that organisations change major database vendors - and if they do, code is likely to be rewritten anyway.

You're here to learn about PL/SQL, so now we know what it is and why you would want to use it, let's get started with the code.

The Basic Structure of a PL/SQL Program

A piece of PL/SQL code is often called a program. A PL/SQL program is structured in blocks. It's a bit different to how functions inside a class work.

A PL/SQL program includes several blocks:

- Declarative section: this is where variables are declared (which you'll learn about later).
- Executable section: this is the code that is run as part of the program goes.
- Exception section: this defines what happens if something goes wrong.

The only required part of a PL/SQL program is the executable section. The other two sections (declarative and exception) are optional.

How do we create these blocks? We use special keywords.

BEGIN and END

The executable part of a PL/SQL program starts with the keyword BEGIN and ends with the keyword END. They are often written on separate lines, like this:

```
BEGIN

--your code goes here

END;
```

The END keyword ends with a semicolon, but the BEGIN keyword doesn't need a semicolon.

The BEGIN keyword starts the executable section of your program. Everything after the BEGIN statement is executed, until the END statement is reached.

This could be your entire PL/SQL program. Just three lines like this. It won't do anything, but it will run.

How can you run PL/SQL code? There are a few places:

- A command line (such as SQL*Plus or SQLcl)
- An IDE (such as SQL Developer or Toad)

Where you run this code depends on how you can access an Oracle database.

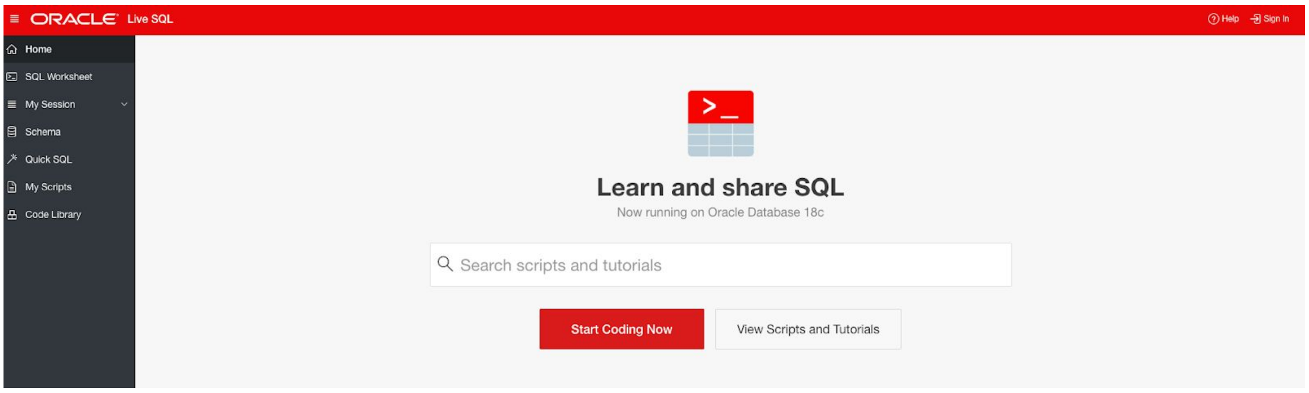
- Are you reading this at work, and have access to an Oracle database for development or testing? Use an IDE that your company or team has available, such as SQL Developer.
- Are you using this at home? Install Oracle Express and SQL Developer (both free) to be able to run code on your own computer.
- Use Oracle LiveSQL, Oracle's free web-based SQL editor.

In this tutorial, we'll be using LiveSQL as it's the easiest to get started on.

Open LiveSQL

Here's how to get started with LiveSQL:

1. Browse to https://livesql.oracle.com



Learn and share SQL

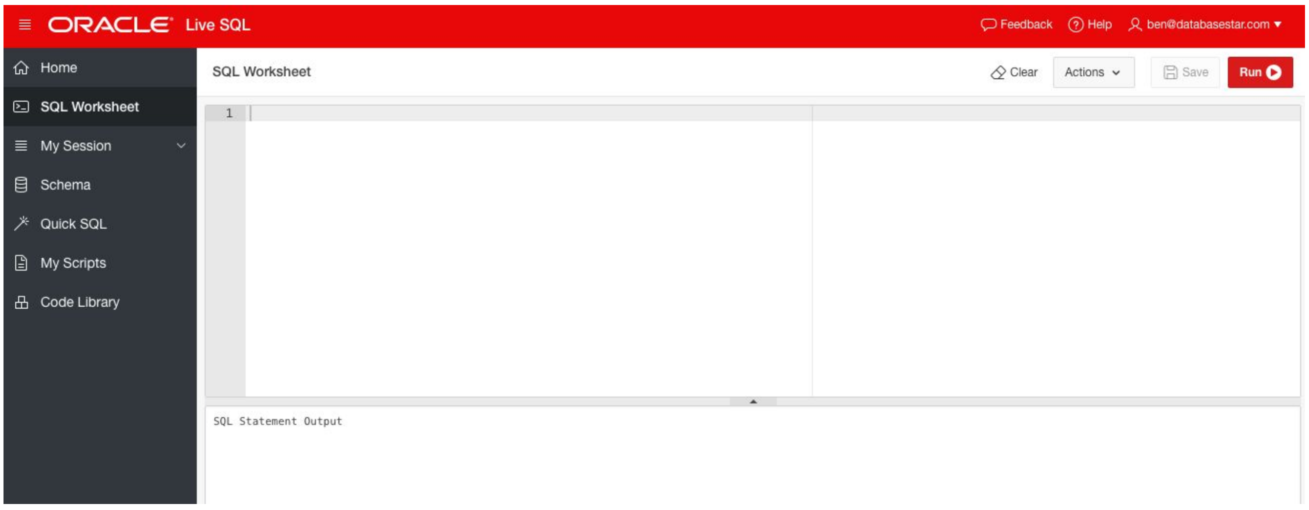
Now running on Oracle Database 18c

2. Click on SQL Worksheet on the left. You'll be taken to the Sign On page.

3. If you have an Oracle account (it's free), enter your username and password. If not, click Create Account.

4. On the Create Account page, fill out the form with your details.

5. Once you have created your account, log in to Live SQL using these details.



The SQL Worksheet is then displayed. This is where you can run SQL statements and see your output.

Now, let's write our first PL/SQL program.

Your First PL/SQL Program: Hello World

If you've learned to program before, you'll probably remember writing your first Hello World program. Hello World is a term in programming where you learn how to write some text to the computer screen in a programming language. The text is often called "Hello World" as a tradition.

So, we'll use PL/SQL code to write the text "Hello World" to the screen.

Enter this code onto the SQL Worksheet:

```
BEGIN


END;
```

Leave a blank line between BEGIN and END. We'll put some code inside there in the moment.

As mentioned before, we'll be using LiveSQL for this tutorial. If you're using an IDE such as SQL Developer, enter your code there.

Now, how to we display data to the screen? We use a function called PUT_LINE.

To call the PUT_LINE function, we need to specify two things:

- The package that the function is contained in
- The text we want to display.

The PUT_LINE function is contained in a package called DBMS_OUTPUT. A package is like a library in other programming languages. It contains a set of related PL/SQL programs. We'll learn about packages and how to create them later in this tutorial.

The text we want to display is "Hello World". This is specified as a parameter, which is inside the brackets that appear after the PUT_LINE function.

So, update your code so it runs the DBMS_OUTPUT.PUT_LINE function in between the BEGIN and END blocks, with the parameter of "Hello World".

The line to include looks like this:

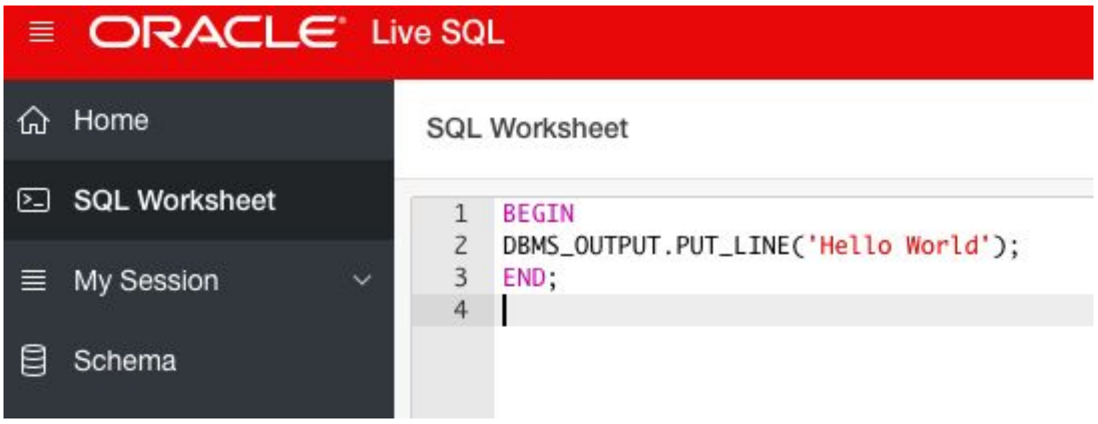
```
DBMS_OUTPUT.PUT_LINE('Hello World');
```

Your program will look like this:

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

The PUT_LINE function includes the text "Hello World" in single quotes, as that's the standard in SQL for working with strings or text values. We also end the line with a semicolon, so the database knows that we have reached the end of the line.

So, your LiveSQL window should look like this:



```
BEGIN
DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

Click the Run button on the top right to run the program. The code will run and display the text in the output section at the bottom of the screen:

In LiveSQL, it shows one line saying "Statement processed", which means the PL/SQL program has run successfully. The second line says "Hello World", which is the text inside your code.

If you're using SQL Developer, you can click the Run button to run the PL/SQL code and you'll see a similar output. If you don't see the output, it's because the DBMS_OUTPUT is disabled by default in SQL Developer. I've written a post with steps and screenshots on how to enable it here. (TODO add link in new tab)

Congratulations! You've just written your first PL/SQL program!

Next, you'll learn about the declarative section in PL/SQL and how to use variables.

Quiz

First, test your knowledge with a quiz.

Question 1:

What does PL/SQL stand for?

- Powerful Loads of Structured Query Language
- Procedural Language Structured Query Language
- Packaged Language Structured Query Language
- Printable Language Structured Query Language

Question 2:

What keyword is used to start the executable section of a PL/SQL program?

- START
- RUN
- BEGIN
- END

Question 3:

What is the built-in function used for displaying output to the screen?

- PRINTLN
- WRITE
- SAVE
- PUT_LINE

Question 4:

What's missing from this simple Hello World program?

```
BEGIN

DBMS_OUTPUT.PUT_LINE

END;
```

- There is no START keyword.
- There is no text value or brackets for the PUT_LINE function.
- Nothing, it will run successfully.
- There is no function called PUT_LINE.

Chapter 2: Variables and Operators

In this chapter, you'll learn:

- What variables are and how to use them in PL/SQL
- Different data types of these variables
- Operators and how PL/SQL handles them

Our Previous Code

The code from the previous lesson looked like this:

```
BEGIN

DBMS_OUTPUT.PUT_LINE('Hello World');

END;
```

It wrote the text "Hello World" to the screen.

What if you wanted to change that text?

You can just change what's inside the function parameter:

```
BEGIN

DBMS_OUTPUT.PUT_LINE('I am learning');

END;
```

This will display the changed text as an output when you run the program.

```
Statement processed.

I am learning
```

One way to improve this code is to store the text we want to output in a variable. PL/SQL supports the use of variables, just like any other programming language.

A variable is a named area in memory that stores the data you tell it to store. They are used to separate the value of something from the treatment of that value. This means you can easily change the value of the output text without altering the function. It also improves readability of the code.

First, we need to learn about the declarative section.

The Declarative Section in PL/SQL

Earlier in this tutorial we mentioned there were several sections of a PL/SQL program:

- Declarative section: this is where variables are declared
- Executable section: this is the code that is run as part of the program goes.
- Exception section: this defines what happens if something goes wrong.

The executable section is defined with BEGIN and END.

The declarative section is defined with the keyword DECLARE. It goes before the BEGIN keyword:

```
DECLARE


BEGIN

DBMS_OUTPUT.PUT_LINE('I am learning');

END;
```

What is the DECLARE section for? We use it to declare variables in PL/SQL. A variable has a name, a data type, and a value.

Let's declare our first variable. We'll call it "l_text". The lowercase L is used as a prefix to indicate it's a local variable (which means it's only relevant for the program we are running). It's not a requirement to add this prefix, but it's recommended by many developers including Steven Feuerstein.

Our code looks like this so far:

```
DECLARE

l_text

BEGIN

DBMS_OUTPUT.PUT_LINE('I am learning');

END;
```

We then have to specify the data type of this l_text variable. All variables need a data type so Oracle knows what kind of data it contains.

We want to store the value that we're going to use in the PUT_LINE function, which is a text value, so we should use the VARCHAR2 data type. Declaring a VARCHAR2 data type is the same in PL/SQL as specifying a column type in SQL, so we need a maximum length. Let's use 50 for this example.

Our code then looks like this:

```
DECLARE

l_text VARCHAR2(50)

BEGIN

DBMS_OUTPUT.PUT_LINE('I am learning');

END;
```

We're almost there. Now, we need to specify what l_text is equal to. We do this by adding a colon, an equals sign, then the new value, then a semicolon. We'll use the value of "Hello World". Our code looks like this:

```
DECLARE

l_text VARCHAR2(50) := 'Hello World';

BEGIN

DBMS_OUTPUT.PUT_LINE('I am learning');

END;
```

Our code now has the variable called l_text, which is of type VARCHAR2 with a maximum of 50 characters. It has been set to the value of "Hello World".

Now, add this code to your SQL worksheet.

Run the code using the Run button. The output looks like this:

```
Statement processed.

I am learning
```

Hang on… we added that line about the variable, but the output still shows the word "I am learning"? How did that happen?

This has happened because even though we added the variable, we didn't change to code to use that variable. The parameter inside the PUT_LINE function still says "I am learning".

Let's change that now. Remove the text "I am learning", and the single quotes, and put the variable name l_text in there instead:

```
DECLARE

l_text VARCHAR2(50) := 'Hello World';

BEGIN

DBMS_OUTPUT.PUT_LINE(l_text);

END;
```

Now, run this code:

```
Statement processed.

Hello World
```

You'll see the words "Hello World" in the output section again. You can now change the value of l_text to whatever you want, and it will be written to the screen.

Why is this useful? It will help us later when we work on more complicate programs, as it improves readability and allows us to manipulate these values in the future.

Indenting

As we add more and more code to our PL/SQL program, it can get harder to read. LiveSQL and many IDEs highlight the keywords and text values in different colours, but there's more we can do to improve readability.

A great way to improve readability is to indent your lines of code. This is commonly done in other programming languages, and is also a good idea in PL/SQL. Whether you use tabs or spaces, indent two, three, or four characters, indenting your code makes it easier to read and work with.

Traditionally, DECLARE, BEGIN, and END keywords are all aligned to the left and not indented. Code in each of those sections is indented. You can do this now with your code:

```
DECLARE

  l_text VARCHAR2(50) := 'Hello World';

BEGIN

  DBMS_OUTPUT.PUT_LINE(l_text);

END;
```

In this tutorial I'll be indenting two spaces, but it's up to you how much you indent. As long as it's consistent!

You'll also notice that I use spaces before and after operators, such as the := symbols. This is not required, but I think it makes the code more readable. You'll find I add spaces in many places in the code for this reason.

Defining Variables Without Assigning Them

In our earlier code, we declared a variable (added a variable and a data type) and assigned a value to it.

However, in PL/SQL, we can declare variables without assigning a value to it at the same time. This is useful if you want to use a variable for something but don't know what the value should be at the time of declaring the variable.

You can declare the name and type of the variable without giving it a value, like this:

```
DECLARE

  l_text VARCHAR2(50);

BEGIN

  DBMS_OUTPUT.PUT_LINE(l_text);

END;
```

As long as the variable has a name and a data type, and ends with a semicolon, it will work. However, if we run this code, this is what we get:

```
Statement processed.
```

There is no output. This is because the variable of l_text is not set, so the PUT_LINE function does not write anything.

We need to set the variable to something inside our program. We can do this by specifying the variable name, then the colon and equals sign, then the variable. This needs to be done inside the BEGIN section of the code:

```
DECLARE

  l_text VARCHAR2(50);

BEGIN

  l_text := 'Hello World';

  DBMS_OUTPUT.PUT_LINE(l_text);

END;
```

This new code has assigned the value of "Hello World" to the variable l_text. The output of this code now looks like this:

```
Statement processed.

Hello World
```

The output now includes the text, because we have assigned it to the variable that was used, before the PUT_LINE statement is run.

If we assign the variable after the PUT_LINE statement, then the output is unchanged, because the PUT_LINE statement is called before the value is changed.

```
DECLARE

  l_text VARCHAR2(50);

BEGIN

  l_text := 'Hello World';

  DBMS_OUTPUT.PUT_LINE(l_text);

  l_text := 'Something else';

END;
```

```
Statement processed.

Hello World
```

So, when you declare variables, you can either assign them at the time you declare them, or assign them later.

Number Data Types

Text values aren't the only variables you can use in PL/SQL. You can use any data type available in Oracle SQL, as well as a few that are PL/SQL specific.

Let's look at some number data types. We can create a variable with the type of NUMBER, and use PUT_LINE to write it to the screen.

This code does exactly that:

```
DECLARE

  l_mynumber NUMBER(8);

BEGIN

  l_mynumber := 12;

  DBMS_OUTPUT.PUT_LINE(l_mynumber);

END;
```

If we run this code, we get this output:

```
Statement processed.

12
```

The PUT_LINE function accepts numbers as well as text. If we want to change the value that is used in this program, we just replace the 12 with our new number.

```
DECLARE

  l_mynumber NUMBER(8);

BEGIN

  l_mynumber := 491;

  DBMS_OUTPUT.PUT_LINE(l_mynumber);

END;

Statement processed.

491
```

Concatenation

In our earlier example, we set the value of a variable, and used that as the output. Displaying the number 12 or 491 wasn't that useful by itself. What if we wanted to add more text to the output? Instead of showing "491", what if we wanted to say "The number you chose was 491"?

We can do that using a feature called concatenation.

What is concatenation?

Concatenation is the ability to join two values together into one. It's often used to combine values into a single output. Many programming languages support it using different characters, such as &, || or a . character. Concatenation can also be done with functions.

In PL/SQL, concatenation is done with a double pipe character ||. You specify a value, then a double pipe, then another value, and the two values are then treated as one. This is helpful in many situations, one of them being the PUT_LINE function.

To add the text "The number you chose was" to the output, we can concatenate that text with the l_mynumber variable:

```
DECLARE

  l_mynumber NUMBER(8);

BEGIN

  l_mynumber := 491;

  DBMS_OUTPUT.PUT_LINE('The number you chose was ' || l_mynumber);

END;
```

Running this code gives us this result:

```
Statement processed.

The number you chose was 491
```

The full output shows the text we added along with the value of the variable.

One thing to keep in mind with concatenation is spaces. I've added a space after the word "was" in the text value. Without the space, the number will be placed right next to the word "was", because concatenation does not automatically add spaces:

```
DECLARE

  l_mynumber NUMBER(8);

BEGIN

  l_mynumber := 491;

  DBMS_OUTPUT.PUT_LINE('The number you chose was' || l_mynumber);

END;

Statement processed.

The number you chose was491
```

Concatenation is very helpful when it comes to working with variables and constructing helpful output, as we've seen above.

Adding Numbers

You can add numbers together in PL/SQL. This is done using the + symbol between two numbers, just like other programming languages and in SQL.

Let's say you wanted to add two numbers together in PL/SQL. Your code could look like this:

```
DECLARE
  l_mynumber NUMBER(8);
BEGIN
  l_mynumber := 4 + 9;
  DBMS_OUTPUT.PUT_LINE('The number you chose was ' || l_mynumber);
END;
```

This code sets the value of l_mynumber to the value of 4 plus 9. This is set to 13, which is shown in the output if you run the code.

```
Statement processed.
The number you chose was 13
```

You can use different number values and add them together in this way.

Subtracting Numbers

If you want to subtract numbers in PL/SQL, you can use the - symbol to subtract one number from the other.

The following code will subtract 5 from the number 100 and display the output.

```
DECLARE
  l_mynumber NUMBER(8);
BEGIN
  l_mynumber := 100 - 5;
  DBMS_OUTPUT.PUT_LINE('The number you chose was ' || l_mynumber);
END;
```

The output looks like this:

```
Statement processed.
The number you chose was 95
```

Multiplying Numbers

You can multiply numbers in PL/SQL by using the * symbol. This is used in many programming languages for multiplication. It's used in SQL for selecting all columns, but if it's used in an expression in SQL or in PL/SQL, it's treated as a multiplication.

The following code will multiple 20 by 4 and display the output.

```
DECLARE

  l_mynumber NUMBER(8);

BEGIN

  l_mynumber := 20 * 4;

  DBMS_OUTPUT.PUT_LINE('The number you chose was ' || l_mynumber);

END;
```

The output looks like this:

```
Statement processed.

The number you chose was 80
```

Dividing Numbers

Finally, PL/SQL allows you to divide numbers by using the / character. The following code will divide 100 by 4 and display the result.

```
DECLARE

  l_mynumber NUMBER(8);

BEGIN

  l_mynumber := 100 / 4;

  DBMS_OUTPUT.PUT_LINE('The number you chose was ' || l_mynumber);

END;
```

The output looks like this:

```
Statement processed.

The number you chose was 25
```

If the number does not divide evenly, the value will still be calculated and displayed, but may not look neat.

```
DECLARE

  l_mynumber NUMBER(8,2)

BEGIN

  l_mynumber := 100 / 7;
  DBMS_OUTPUT.PUT_LINE('The number you chose was ' || l_mynumber);
END;
```

The output looks like this:

```
Statement processed.
The number you chose was 14
```

The reason this shows 14 is because the data type is declared as an 8-digit number. No precision is specified so it uses a precision of 0, rounding down to the nearest number. It shows 100 divided by 7 which is 14.29, rounded down to 14.

Constants

In the code so far, we've declared variables and assigned them. We've also declared variables and assigned them later in the PL/SQL program. This is an advantage of using variables - you can update the value of them in your program.

For example, let's say you had this PL/SQL program which calculated the circumference of a circle. The circumference of a circle is 2πr, or 2 times the radius times pi. The value of pi is approximately equal to 3.14159.

Our code may look like this:

```
DECLARE
  l_radius NUMBER(4, 3);
BEGIN
  l_radius := 8;
  DBMS_OUTPUT.PUT_LINE('The circumference is ' || 2 * l_radius *
3.14159);
END;
```

Our output looks like this:

```
Statement processed.
The circumference is 50.26544
```

This code includes a variable called l_radius which is the radius of our circle. We set this value to 8 inside our code, and then output the value of the circumference as 2 * l_radius * 3.14159.

Let's use our knowledge of variables to declare a variable for the circumference, set that to the calculated value, and use that variable in the output.

```
DECLARE
  l_radius NUMBER(4, 3);
  l_circumference NUMBER(10, 3);
```

```
BEGIN
  l_radius := 8;
  l_circumference := 2 * l_radius * 3.14159;
  DBMS_OUTPUT.PUT_LINE('The circumference is ' || l_circumference);
END;
```

Our output looks like this:

```
Statement processed.
The circumference is 50.265
```

This is good, but looking at this code you might wonder what the significance of the number 3.14159 is. This example may make sense as you might know what pi is equal to, but the code should actually explain this without you needing to know it.

So, let's move that number into a variable. We'll set it when we declare it because we don't need to change it.

```
DECLARE
  l_radius NUMBER(4, 3);
  l_circumference NUMBER(10, 3);
  l_pi NUMBER(6, 5) := 3.14159;
BEGIN
  l_radius := 8;
  l_circumference := 2 * l_radius * l_pi;
  DBMS_OUTPUT.PUT_LINE('The circumference is ' || l_circumference);
END;
```

The output of this code is:

```
Statement processed.
The circumference is 50.265
```

The code runs, we've used variables, and it's easy to read.

What if, for some reason, we change the value of pi within our code?

```
DECLARE
  l_radius NUMBER(4, 3);
  l_circumference NUMBER(10, 3);
  l_pi NUMBER(6, 5) := 3.14159;
BEGIN
```

```
  l_radius := 8;
  l_pi := 4;
  l_circumference := 2 * l_radius * l_pi;
  DBMS_OUTPUT.PUT_LINE('The circumference is ' || l_circumference);
END;
```

We can run our code and get this output.

```
Statement processed.
The circumference is 64
```

The output shows a value of 64, which is not correct according to our rules. This is because we have set pi to 4.

Is there any way to prevent this from being changed in our code after we specify it?

Yes, there is. We can declare the l_pi variable as a constant. A constant is a type of variable that is not allowed to change during the program. Once it is declared and assigned, it can't be changed. It's great for values that never change, such as pi, and other values your program uses that don't change. It helps prevent errors such as this.

To declare a variable as a constant, we add the keyword CONSTANT after the variable name and before the data type. The variable also needs to be defined and assigned at the same time.

To set the variable of l_pi as a constant, your code will look like this:

```
DECLARE
  l_radius NUMBER(4, 3);
  l_circumference NUMBER(10, 3);
  l_pi CONSTANT NUMBER(6, 5) := 3.14159;
BEGIN
  l_radius := 8;
  l_circumference := 2 * l_radius * l_pi;
  DBMS_OUTPUT.PUT_LINE('The circumference is ' || l_circumference);
END;
```

This variable is now a constant. This code will run and show the right output:

```
Statement processed.
The circumference is 50.265
```

If you try to change the value of l_pi during the program, you'll get an error.

```
DECLARE
```

```
  l_radius NUMBER(4, 3);
  l_circumference NUMBER(10, 3);
  l_pi CONSTANT NUMBER(6, 5) := 3.14159;
BEGIN
  l_radius := 8;
  L_pi := 4;
  l_circumference := 2 * l_radius * l_pi;
  DBMS_OUTPUT.PUT_LINE('The circumference is ' || l_circumference);
END;
```

```
ORA-06550: line 7, column 3: PLS-00353: expression 'L_PI' cannot be used
as an assignment target
```

This error means you can't assign a value to this variable, because you've declared it as a constant. This is what we expected.

Conclusion

In this chapter, you learned:

- What variables are, how to declare them, and how to assign them
- Using text and number variables
- How to concatenate two variables together
- How to add, subtract, multiply, and divide numbers
- How to declare constants

Well done for getting this far! In the next chapter, you'll learn about some key features of PL/SQL: conditional logic and loops.

Quiz

To test your PL/SQL knowledge so far, take this quiz on the topics covered in this chapter.

Question 1:

What keyword is used to start the section where variables are created?

- START
- BEGIN

- DECLARE
- VARIABLE

Question 2:

What do you need to include when declaring a variable?

- The name of the variable and the data type are required.
- The name, data type, and the value are all required
- The name of the variable is required.
- The name, data type, the value, and the word CONSTANT are all required.

Question 3:

What is a constant?

- An area in memory that stores a value for your program and can be changed in your program.
- A variable in your program that cannot be changed once it is set.
- A function that displays data to the screen.
- A way to join two values together into one.

Question 4:

What's wrong with this code?

```
DECLARE
  l_radius NUMBER(8);
BEGIN
  DBMS_OUTPUT.PUT_LINE('The diameter is ' || 2 * l_radius);
END;
```

- Nothing. It will run and display the value of 16, because l_radius is set to 8 and the function multiplies it by 2.
- The code will run, but it won't display a number because the value of l_radius is not set.
- The code will run and display a value of 2.
- The code will not run because you can't concatenate a text value with a number value.

Chapter 3: Conditions and Loops

In this chapter, you'll learn about:

- Conditions (If, Then, Else)
- Loops
- Commenting Code

These are common features of other programming languages, and PL/SQL lets you use these features as well.

What Are Conditions?

In the code we have written so far, the database executes every line of code from top to bottom. This is acceptable for simple programs. But as you learn more and write more code, you'll eventually want the code to do different things based on different conditions, such as:

- Display a message if a user's account balance is below a certain number
- Perform a different action depending on the account type
- Display a different value depending on the location of a user

Each of these can be done by using PL/SQL keywords that allow conditions. Conditions are where you run different lines of code based on a specific condition. You can specify the condition to check, and this check will return true or false. If it's true, then a set of code is run.

A Simple IF Statement

An IF statement is where a condition is checked, and if it is true, a set of code is run.

This can be represented in PL/SQL code in this way:

```
IF (condition) THEN

  your_code;

END IF;
```

This looks similar in many programming languages, and the only differences are usually the keywords and symbols used.

Let's take a look at an example. We want to find the length and width of a rectangle, and if they are the same, we will display a message saying it is a square.

```
DECLARE

  l_width NUMBER(5) := 20;
```

```
 l_length NUMBER(5) := 20;

BEGIN

  IF (l_width = l_length) THEN

    DBMS_OUTPUT.PUT_LINE('This is a square.');

  END IF;

END;
```

We have two variables: one for the length and one for the width. Inside the BEGIN block, we have an IF statement.

Inside the brackets of the IF statement is the condition. The condition checks that the width equals the length, which is done using the = sign. If this calculation is true, then the code inside runs, which is the PUT_LINE function.

The IF statements ends with the END IF. We then finish the code with the END statement. Also notice the lines that have a semicolon: the put_line function, the END IF, and the END statement.

If we run this code, this is output:

```
Statement processed.

This is a square.
```

This message is shown because the length and width are equal. We can change the values of the variables and see if the output changes.

```
DECLARE

  l_width NUMBER(5) := 20;

 l_length NUMBER(5) := 20;

BEGIN

  IF (l_width = l_length) THEN

    DBMS_OUTPUT.PUT_LINE('This is a square.');

  END IF;

END;
```

```
Statement processed.
```

Notice that the output does not say it is a square. This is because the IF statement does not return a true value, because the length and width are different (18 and 20).

That's a simple IF statement in PL/SQL: code that runs if a condition is true.

Running Code When False with ELSE

In the earlier example, we had some PL/SQL code that displayed a message if the values were the same, and therefore the shape is a square. Nothing was displayed if the values were not the same.

What if we wanted to do something if these values were not the same, or if the condition was false?

We can do that using the ELSE statement. The ELSE statement runs code if the condition is false. In a diagram, it looks like this.

This can be represented in PL/SQL code in this way:

```
IF (condition) THEN

  your_code;

ELSE

  your_other_code;

END IF;
```

You simply add the ELSE keyword after the code you run if the condition is true, then add code you want to run if the condition is false.

Let's say we wanted to expand on the earlier example to show a message if the length and width is not equal. We will display a message saying the shape is a rectangle.

Our code would look like this:

```
DECLARE

  l_width NUMBER(5) := 20;

  l_length NUMBER(5) := 20;

BEGIN

  IF (l_width = l_length) THEN

    DBMS_OUTPUT.PUT_LINE('This is a square.');

  ELSE

    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');

  END IF;

END;
```

The only changes to this code are adding the ELSE statement and a put_line function to say "This is a rectangle".

The output of this code is:

```
Statement processed.

This is a square.
```

This output is still showing the message of "square" because both the length and width variables are the same. We can change one and the output will change.

```
DECLARE
  l_width NUMBER(5) := 17;
  l_length NUMBER(5) := 20;
BEGIN
  IF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

```
Statement processed.
This is a rectangle.
```

The code has determined that the length and width (17 and 20) are different, so it has run the code after the ELSE statement and not the IF statement.

So that's how you can perform a simple IF statement and run code if the condition is true and different code if the condition is false.

Checking Multiple Conditions with ELSIF

Another feature of PL/SQL IF statements is the ability to have multiple conditions inside a single IF statement.

Using our earlier example, we check if the length and width are the same, display "square" if they are and "rectangle" if they are not. What if we want to add another check to see if the width is less than 0?

Our code so far hasn't included any logic for this. If both numbers are negative, then the code will still run:

```
DECLARE
  l_width NUMBER(5) := -4;
  l_length NUMBER(5) := -3;
BEGIN
  IF (l_width = l_length) THEN
```

```
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;


Statement processed.
This is a rectangle.
```

We want to do something if the width value is negative. We would need to add another check inside this statement. We can do that using the ELSIF keyword, which is short for "else if".

In PL/SQL code, the code looks like this:

```
IF (condition) THEN
  your_code;
ELSIF (condition2) THEN
  your_second_code
ELSE
  your_other_code;
END IF;
```

The ELSIF goes after the code that is run for the IF statement, and before the ELSE statement. If the first IF condition is false, then the condition inside the ELSIF is checked. If that is true, then the code inside the ELSIF is run, otherwise the ELSE statement is run.

Let's expand our example to check if the width is negative and display a message.

```
DECLARE
  l_width NUMBER(5) := -4;
  l_length NUMBER(5) := -3;
BEGIN
  IF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSIF (l_width < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width is negative.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;

END;
```

The check of l_width < 0 will return a value of true if the l_width is negative, and the message below will be displayed. This code will only be run if the width and length are not equal.

The output is:

```
Statement processed.

The width is negative.
```

You can use the ELSIF keyword to add extra conditions to your IF statement.

What if both the width and length are negative? Which condition will be checked? They are equal and negative so they meet both conditions. Let's test this out.

```
DECLARE

  l_width NUMBER(5) := -4;

  l_length NUMBER(5) := -3;

BEGIN

  IF (l_width = l_length) THEN

    DBMS_OUTPUT.PUT_LINE('This is a square.');

  ELSIF (l_width < 0) THEN

    DBMS_OUTPUT.PUT_LINE('The width is negative.');

  ELSE

    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');

  END IF;

END;


Statement processed.

This is a square.
```

The output says it is a square. This is because the first IF statement ran, found that the values are equal, displayed the message, and then ended the IF statement. If one condition is true, the rest are skipped: they aren't run. This is an important point to keep in mind.

This could be a problem in your application. If you would prefer to display a message about the number being negative rather than it being a square, you can move your conditions around, and place the negative check above the square check:

```
DECLARE

  l_width NUMBER(5) := -4;
  l_length NUMBER(5) := -4;
BEGIN
  IF (l_width < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width is negative.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

The check for the width being negative comes first, and then if that is false, the length and width are checked to see if they are equal.

The output of this code is shown below:

```
Statement processed.
The width is negative.
```

We can see the output makes more sense now. The values are equal, but because the width is negative, it displays the negative message.

Multiple Criteria In A Condition

There was another issue in our code above. We have checked the width to see if it is negative, but what if the width is positive and the length is negative?

```
DECLARE
  l_width NUMBER(5) := 6;
  l_length NUMBER(5) := -4;
BEGIN
  IF (l_width < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width is negative.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

This code shows the following output:

```
Statement processed.
This is a rectangle.
```

We don't want the message to say it is a rectangle. We would rather a message to say that one of the numbers is negative.

Rather than adding another IF condition, we can adjust the one we already have. PL/SQL allows you to add multiple criteria inside an IF statement condition.

For example, we could check if either the length or width is negative, and if so, display a message saying that one of the values is negative. Our code would look like this:

```
DECLARE
  l_width NUMBER(5) := 6;
  l_length NUMBER(5) := -4;
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

Notice that our first IF statement includes this code:

```
l_width < 0 OR l_length < 0
```

The two conditions have been separated by the OR keyword. This means that if either the width < 0 or the length < 0 then the IF statement will return true, and the code underneath will run. This will mean that our example should show the right message:

```
Statement processed.
The width or length is negative.
```

This will handle situations where one of the values is negative and the other is positive.

You can also use the AND keyword to check that both conditions are true. For example, let's say you wanted to display a separate message for "big squares" and a big square is where the length is greater than 50 and the length and width are the same. Your code would look like this:

```
DECLARE
  l_width NUMBER(5) := 55;
  l_length NUMBER(5) := 55;
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

The code shows this output:

```
Statement processed.
This is a big square.
```

This code shows that the specified values make a big square.

Reversing a Condition

So far we've looked at code that checks that conditions are true. What if we want to check if conditions are false?

Generally, it's better practice to check if conditions are true, and it makes for easier reading of code. However, sometimes you may need to check if something is false. This can be done with the NOT keyword.

Let's take our earlier example and change the condition from checking if the width and length are less than zero to checking if the width and length are not greater than zero

```
DECLARE
  l_width NUMBER(5) := -3;
  l_length NUMBER(5) := -2;
BEGIN
  IF NOT (l_width > 0 AND l_length > 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

The changed line of code is this one:

```
IF NOT (l_width > 0 AND l_length > 0) THEN
```

We have added the word NOT outside the brackets, which means the condition inside the brackets needs to be false for the IF statement to be true. The NOT keyword has reversed the logic.

Why did we also change the OR to AND? Because of how they work.

We could have left it like this:

```
IF NOT (l_width > 0 OR l_length > 0) THEN
```

If we did, then it will only return TRUE if both of the values are false. It can be explained by this table:

l_width	**l_length**	**Length > 0**	**Width > 0**	**Using OR**	**Using AND**
-3	-2	FALSE	FALSE	FALSE	FALSE
4	-2	TRUE	FALSE	TRUE	FALSE
-3	5	FALSE	TRUE	TRUE	FALSE
4	5	TRUE	TRUE	TRUE	TRUE

The only situation where both of the numbers are positive is where AND returns true. If we use OR, the statement returns TRUE if either of the numbers are positive. This can result in the wrong message being shown in our code.

Nested IF Statements

We've just seen how you can use IF statements in PL/SQL. This allows you to check conditions and run different code depending on these conditions.

PL/SQL allows for IF statements to be nested inside other IF statements, if that's something you need to do to implement the logic you want.

The code would look like this:

```
IF (condition) THEN
  IF (condition) THEN
    your_code;
  ELSE
    more_code;
  END IF;
ELSIF (condition2) THEN
  your_second_code
ELSE
  your_other_code;
END IF;
```

You add in another IF statement where your code would normally go. You can use ELSE and ELSIF statements there as well.

PL/SQL supports a large number of nested IF statements, so you can nest as many as you need. However, if you start getting considering the maximum number of nesting while you're programming, you probably need to rewrite your code. Nesting more than 4 or 5 levels deep is usually not the best way to structure your code.

Let's see an example of a nested IF statement. We'll adjust our earlier code to write a message that lets us know which of the values is negative.

```
DECLARE
  l_width NUMBER(5) := -3;
  l_length NUMBER(5) := -2;
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    IF (l_width < 0) THEN
      DBMS_OUTPUT.PUT_LINE('The width is negative.');
    END IF;
    IF (l_length < 0) THEN
      DBMS_OUTPUT.PUT_LINE('The length is negative.');
    END IF;
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

This code has some nested IF statements inside the IF statement that checks for the width or length being less than 0.

This shows the following output:

```
Statement processed.
The width is negative.
The length is negative.
```

Two lines of output have been written, because both of the nested IF statements are true. This shows how you can add an IF statement inside another IF statement.

The CASE Statement

IF statements are powerful. They let you check for different conditions and run different code. Your program might get to a point where there are several IF and ELSIF statements. If so, it can start to look a little messy.

You might be wondering, is there a better way to do this?

Yes, there is. It's called a CASE statement.

A CASE statement lets you specify one or more conditions, and if the condition is true, then some code is run. It's easier to write and read than a long IF statement.

The CASE statement looks like this:

```
CASE condition
WHEN value THEN code;
WHEN value THEN code;
...
ELSE default_code;
END CASE;
```

It looks and operates the same as a CASE statement in regular SQL.

There are a few things in this CASE statement:

- A condition: this is the condition that is checked.
- WHEN value: this defines the value of the condition that is used in the following code.
- THEN code: this code is run if the value is equal to the condition.
- ELSE default_code: this code is run if none of the specified values match the condition.

Let's see an example. Let's say that we wanted to show specific messages if the length is one of a few different values.

```
DECLARE
  l_width NUMBER(5) := 5;
  l_length NUMBER(5) := 10;
BEGIN
  CASE l_length
    WHEN 10 THEN
      DBMS_OUTPUT.PUT_LINE('The length is ten.');
    WHEN 20 THEN
      DBMS_OUTPUT.PUT_LINE('The length is twenty.');
    WHEN 30 THEN
      DBMS_OUTPUT.PUT_LINE('The length is thirty.');
    ELSE
      DBMS_OUTPUT.PUT_LINE('The length is another value.');
  END CASE;
END;
```

This code will check the l_length value, and display a specific output if it is equal to 10, 20, or 30. If it's different, it displays a different message.

If we run the code, this is what we see:

```
Statement processed.
```

```
The length is ten.
```

The output shows the message "The length is ten". If we change the length to 30, this is what we see.

```
Statement processed.

The length is thirty.
```

If the length is not equal to any of the values in the CASE statement, we see the other message.

```
DECLARE
  l_width NUMBER(5) := 5;
  l_length NUMBER(5) := 10;
BEGIN
  CASE l_length
    WHEN 10 THEN
      DBMS_OUTPUT.PUT_LINE('The length is ten.');
    WHEN 20 THEN
      DBMS_OUTPUT.PUT_LINE('The length is twenty.');
    WHEN 30 THEN
      DBMS_OUTPUT.PUT_LINE('The length is thirty.');
    ELSE
      DBMS_OUTPUT.PUT_LINE('The length is another value.');
  END CASE;
END;
```

```
Statement processed.

The length is another value.
```

Only one of the conditions in the CASE statement is calculated. Unlike other programming languages, we don't need to add a break statement to exit this code. As soon as the first criteria is met, it exits the case statement. If no matches are found, the ELSE statement is run.

What is a Loop?

We've just learned about IF statements and CASE statements, which let you run different pieces of code depending on different conditions.

Another useful feature of PL/SQL is a loop.

What is a loop?

A loop is a feature of programming languages that lets you run the same piece of code multiple times. You can specify the conditions that ensure the loop is run, such as the maximum number of iterations of the loop, or until a certain condition is met.

There are several ways to write loops in PL/SQL, which we will explain in this guide.

A Basic Loop

The most basic kind of loop in PL/SQL will start a loop (a set of code that is executed multiple times). It will only exit when you tell it to. It uses the following keywords:

- LOOP to start the loop
- END LOOP to define the end of the code that's run as part of the loop
- EXIT to stop running the loop

The code would look something like this:

```
BEGIN

  LOOP

    your_code

    IF (condition) THEN

      EXIT;

    END IF;

  END LOOP;

END;
```

The LOOP and END LOOP indicate that all code inside it should be run multiple times. The code inside the loop is executed, line by line. When the END LOOP is reached, the code starts again from the LOOP statement.

If an EXIT statement is reached, the loop exits, and any code after the loop is then run.

Let's see an example, where we want to display an output five times on the screen.

Our code would look like this:

```
BEGIN

  LOOP

    DBMS_OUTPUT.PUT_LINE('A message here.');

  END LOOP;

END;
```

If we run this code, it will keep running without stopping unless you force it to stop in your IDE (or reset your session in LiveSQL).

This is called an infinite loop. It happens because the code is looping and you haven't told it to stop. This is an issue that happens occasionally in code, and is something to be avoided.

So how to we stop this loop from running forever? We need to tell it to exit at some point.

```
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('A message here.');
    IF (condition) THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

But what's our condition? We want to run the loop and display the message five times, so we need to do two things:

- Keep track of how many times the loop has run
- Check if the loop has run five times each time it runs

We can do this using a variable:

```
DECLARE
  l_loop_counter NUMBER(3) := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('A message here.');
    IF (condition) THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

We have declared the variable l_loop_counter and assigned it a value of 0, which indicates the number of times the loop has been run. We now need to increase the value by 1 each time we run the loop.

```
DECLARE
  l_loop_counter NUMBER(3) := 0;
```

```
BEGIN
  LOOP
    l_loop_counter := l_loop_counter + 1;
    DBMS_OUTPUT.PUT_LINE('A message here.');
    IF (condition) THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

This code will increase that variable by 1 each time the loop is run. We then need to add this to our condition, so the loop exits when we want it to.

```
DECLARE
  l_loop_counter NUMBER(3) := 0;
BEGIN
  LOOP
    l_loop_counter := l_loop_counter + 1;
    DBMS_OUTPUT.PUT_LINE('A message here.');
    IF (l_loop_counter = 5) THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

This code now says the loop will exit when the l_loop_counter value equals 5. Let's run this code:

```
Statement processed.
A message here.
A message here.
A message here.
A message here.
A message here.
```

The output shows the line "A message here" five times, which meets the criteria of our code. The loop counter starts at 0, is incremented by 1 each time it is run, and once it gets to 5 the loop exits.

You can change the value inside the condition from 5 to 10 or another number to change the number of times the loop is run.

```
Statement processed.

A message here.

A message here.

A message here.

A message here.

A message here.

A message here.

A message here.

A message here.

A message here.

A message here.
```

What if you wanted to show the number of the loop in each message? You can do that using concatenation which we learned about earlier. You can concatenate the loop counter variable to the message:

```
DECLARE

  l_loop_counter NUMBER(3) := 0;

BEGIN

  LOOP

    l_loop_counter := l_loop_counter + 1;

    DBMS_OUTPUT.PUT_LINE('A message here: ' || l_loop_counter);

    IF (l_loop_counter = 5) THEN

      EXIT;

    END IF;

  END LOOP;

END;
```

This code shows the loop counter after the message:

```
Statement processed.

A message here: 1

A message here: 2

A message here: 3
```

```
A message here: 4

A message here: 5
```

Using different features of PL/SQL means you can create the programs you really need.

The For Loop

PL/SQL offers another type of loop called the FOR loop. This FOR loop allows you to define the criteria of the loop at the start, which makes it easier to see how the loop runs and easier to avoid "infinite loop" issues.

The syntax of a FOR loop looks like this:

```
FOR counter IN start_value ..  end_value LOOP

  your_code

END LOOP;
```

This includes a few things:

- The FOR keyword starts the loop
- The counter variable is the variable that is used as an counter for the loop. It is incremented/increased by 1 each time the loop is run.
- The start_value is the value that the counter is initially set to.
- The end_value is the value that the counter ends on. When the counter value is greater than the end_value, the loop exits and the code after the loop is run.

An example of the PL/SQL FOR loop is shown here.

```
DECLARE

  l_loop_counter NUMBER(3);

BEGIN

  FOR l_loop_counter IN 1 .. 5 LOOP

    DBMS_OUTPUT.PUT_LINE('Loop number: ' || l_loop_counter);

  END LOOP;

END;
```

This code is much shorter. There is no need for an IF statement to check if the counter is still in range. There is no need for a separate line to increase the value of the loop counter, as it's done as part of the FOR loop.

Also, notice the syntax of the start and end value:

```
1 .. 5
```

It's a number, then a space, then two periods, then a space, then another number. If it's not written in this way, you'll get a syntax error. I've done this many times - forgotten the space, only put in one period - and wondered why the code didn't run. So make sure you check that when you write your code.

If we run this code, we get this result:

```
Statement processed.

Loop number: 1

Loop number: 2

Loop number: 3

Loop number: 4

Loop number: 5
```

We can see the output is similar, and the code is much shorter. FOR loops are great in this way: they include a lot of the parts of a normal LOOP in the syntax, which means less code.

You can change the start and ending values of the loop counter to whatever numbers you want. But the increment on each loop can only go up or down by 1:

```
DECLARE

  l_loop_counter NUMBER(3);

BEGIN

  FOR l_loop_counter IN 12 .. 20 LOOP

    DBMS_OUTPUT.PUT_LINE('Loop number: ' || l_loop_counter);

  END LOOP;

END;


Statement processed.

Loop number: 12

Loop number: 13

Loop number: 14

Loop number: 15

Loop number: 16

Loop number: 17

Loop number: 18
Loop number: 19

Loop number: 20
```

As you can see, the FOR loop is very useful. I generally prefer using it to the regular LOOP syntax.

The While Loop

Another type of loop offered in PL/SQL is the WHILE loop.

What is the WHILE loop?

It's a type of loop that runs until the specified condition is met. It's similar to the basic LOOP but includes a condition when you define it:

```
WHILE (condition) LOOP

  your_code

END LOOP;
```

This loop syntax includes a condition, which means the code inside the loop only runs if the condition is true. The code inside the loop must ensure that the condition eventually results in FALSE, otherwise you'll end up with an infinite loop.

For example, this code will cause an infinite loop.

```
DECLARE

  l_loop_counter NUMBER(3);

BEGIN

   WHILE (l_loop_counter < 5) LOOP

    DBMS_OUTPUT.PUT_LINE('Loop number: ' || l_loop_counter);

  END LOOP;

END;
```

This code will cause an infinite loop because there is no code that increases the l_loop_counter or that initialises it with a value. We need to add this code manually.

```
DECLARE

  l_loop_counter NUMBER(3) := 0;

BEGIN

   WHILE (l_loop_counter < 5) LOOP

    DBMS_OUTPUT.PUT_LINE('Loop number: ' || l_loop_counter);

    l_loop_counter := l_loop_counter + 1;
  END LOOP;
END;
Statement processed.
Loop number: 0
Loop number: 1
Loop number: 2
Loop number: 3
Loop number: 4
```

The output shows 5 rows as expected. The values are 0 to 4 because the loop counter starts at zero and keeps going until it is not less than 5.

Differences Between Loop Types

As you can see, there are three different types of loops in PL/SQL. There are some slight differences between each of them though, which are summarised in this table:

Criteria	**Basic LOOP**	**FOR**	**WHILE**
Includes the counter definition	No	Yes	No
Includes the exit criteria	No	Yes	Yes
Increments a counter	No	Yes	No
Runs the first line of the loop	Yes	Yes	Not always

One of the main differences between the WHILE and FOR loops is that the WHILE loop may not always run the code inside the loop. If the condition used in the WHILE loop is false when it is first checked, the code inside the loop is not run.

Conclusion

The PL/SQL language includes powerful features for running certain pieces of code in certain conditions. This includes IF THEN ELSE logic, CASE statements, and looping. There are three types of loops (Basic, FOR, and WHILE), each of them are slightly different.

Quiz

Question 1:

What is the keyword used as part of an IF statement that runs code if none of the provided conditions are met?

- ELSE
- OTHERWISE
- FINAL
- END

Question 2:

What does the FOR statement do?

- Initialises a variable
- Checks if a condition is true or false, and runs the code inside one time if it is true
- Starts a loop and runs the code inside many times until the criteria is false
- Sets up the data required to send an email

Question 3:

Which of the following pieces of code will correctly check that two conditions are true in an IF statement?

- IF (l_input > 10) (l_input < 20) THEN
- IF (l_input > 10 & l_input < 20) THEN
- IF (l_input > 10 < 20) THEN
- IF (l_input > 10 AND l_input < 20) THEN

Question 4:

What is wrong with this code?

```
DECLARE
  l_loop_counter NUMBER(3) := 8;
BEGIN
   WHILE (l_loop_counter < 5) LOOP
    DBMS_OUTPUT.PUT_LINE('Loop number: ' || l_loop_counter);
    l_loop_counter := l_loop_counter + 1;
  END LOOP;
```

```
END;
```

- It will run but no output is shown.
- Nothing, it will run successfully and show five lines of output.
- It will run but cause an infinite loop.
- It will run and show eight lines of output.

Chapter 4: Procedures, Functions, and Exceptions

In this chapter, you'll learn:

- What procedures and functions are
- How to create a procedure
- How to create a function
- What an exception is and how to handle one

Let's get started!

What is a PL/SQL Procedure?

A PL/SQL procedure, or stored procedure, is a set of code stored on the database and has a specific name. This procedure can be run by calling it in other code using this name, and the code inside the procedure will run and perform the actions inside.

Why create stored procedures?

- It's where most of your PL/SQL code will go, rather than writing the unnamed code like we have done so far.
- It allows for the code to be reused by calling it from other code
- It allows for an interface or API to the database.

Earlier in this guide I mentioned one of the benefits of PL/SQL was to provide a common interface to the database. Procedures let you do that.

For example, rather than relying on every application or developer trying to write an INSERT statement that works, you can create a procedure that accepts a few parameters and runs an INSERT statement with them. This means the INSERT statements are consistent and the developers know what they need to provide.

What is a PL/SQL Function?

A PL/SQL function is a named piece of code on the database that lets you provide parameters, perform a calculation, and return a result. They can be used in PL/SQL or in regular SQL. PL/SQL functions are similar to procedures but have some differences.

What's the Difference Between a PL/SQL Procedure and Function?

The main differences between a PL/SQL procedure and a PL/SQL function are:

- Functions must return a value. Procedures don't return values but can use OUT parameters.

- Functions can be called from regular SQL, but procedures cannot.

Those are the main two differences. Procedures and functions are often used for two different things:

- Performing calculations and returning a value? Use a function.
- Performing steps and inserting, updating, or deleting database records? Use a procedure.
- Need to call the code from SQL? Use a function.

Let's take a look at how you can create a procedure and a function.

How to Create a PL/SQL Procedure

A PL/SQL procedure is created with the CREATE PROCEDURE statement. The syntax of the statement looks like this:

```
CREATE [OR REPLACE] PROCEDURE procedure_name

[ ( parameter_name [ IN | OUT | IN OUT ] parameter_type [, ...] ) ]

{ IS | AS }

BEGIN

  procedure_body

END procedure_name;
```

There are a few things to notice here:

- The procedure_name is the name of the procedure to create.
- The OR REPLACE keyword is optional. If you include it, it means that when you run this statement it will replace the procedure with the same name. If you don't include it and a procedure with this name already exists, you'll get an error.

One or more parameters can be provided:

- Parameter_name is the name of the parameter, which is used in your procedure code
- A parameter can be an IN parameter (value is provided to the procedure to use), an OUT parameter (value is determined by the procedure and returned to the code that has called it), or IN OUT (combination of IN and OUT).
- A parameter has a parameter_type which is an SQL data type, such as NUMBER.
- Further parameters can be added and separated by commas.

After you define the parameters:

- Specify either IS or AS to start the procedure code.
- BEGIN is specified just like we have learned so far.
- You add the code for your procedure, and finish with END procedure_name.

Example PL/SQL Procedure

Let's take a look at an example. We'll use our earlier code that checks values for a square or rectangle:

```
DECLARE
  l_width NUMBER(5) := 18;
  l_length NUMBER(5) := 15;
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END;
```

This code is not a procedure or a function, as it has no name. This code is called an "anonymous block", because it has no name (it's anonymous) and is a block of code. So, if anyone refers to an anonymous block, it means a section of PL/SQL code that is not a function or procedure.

To convert this to a procedure, we need to give it a name. Let's call it "check_rectangle". Names of functions and procedures in PL/SQL must meet the following criteria:

- The maximum length is 30 characters
- The first character must be a letter
- The characters after the first character can be letter, number, dollar sign $, number sign #, or an underscore _.

Also, PL/SQL names are case-insensitive. This means the following names are all equivalent:

- check_rectangle
- CHECK_RECTANGLE
- Check_Rectangle

It's up to you what case you choose for your PL/SQL names, as it comes down to your coding standards. In this guide, I'll be using all lowercase for my procedure and function names.

Let's start our check_rectangle procedure:

```
CREATE OR REPLACE PROCEDURE check_rectangle
```

I have added the OR REPLACE keywords here so that you can run the same statement over and over again if you want to make any changes. You don't need to drop the procedure separately.

We then have either the IS or AS keyword. I'll use the AS because it's a personal preference. There is no difference between these two keywords.

```
CREATE OR REPLACE PROCEDURE check_rectangle

AS
```

Then we can add in our code. We can copy and paste the code from earlier (the anonymous block) and add it here, and add the procedure_name to the END keyword.

```
CREATE OR REPLACE PROCEDURE check_rectangle

AS

DECLARE

  l_width NUMBER(5) := 18;

  l_length NUMBER(5) := 15;

BEGIN

  IF (l_width < 0 OR l_length < 0) THEN

    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');

  ELSIF (l_width = l_length AND l_length > 50 ) THEN

    DBMS_OUTPUT.PUT_LINE('This is a big square.');

  ELSIF (l_width = l_length) THEN

    DBMS_OUTPUT.PUT_LINE('This is a square.');

  ELSE

    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');

  END IF;

END check_rectangle;
```

If we run this code, we get an error message:

```
Errors: PROCEDURE CHECK_RECTANGLE Line/Col: 3/1 PLS-00103: Encountered
the symbol "DECLARE" when expecting one of the following: begin function
pragma procedure subtype type <an identifier> <a double-quoted
```

```
delimited-identifier> current cursor delete exists prior external
language
```

This error appears because we have a DECLARE statement in our code. Because we're writing a procedure, we don't need the DECLARE keyword. The variables can go just after the AS or IS keyword and before the BEGIN. Simply remove the DECLARE keyword:

```
CREATE OR REPLACE PROCEDURE check_rectangle
AS
  l_width NUMBER(5) := 18;
  l_length NUMBER(5) := 15;
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END check_rectangle;
```

You'll get this output:

```
Procedure created.
```

When creating a procedure like this, the procedure is only created and not run. You'll only see the "Procedure created" message, not any of the output inside the procedure. You'll need to call or run the procedure to do that.

To run or call a PL/SQL stored procedure, you can do one of three things:

- CALL procedure_name;
- EXEC procedure_name;
- BEGIN procedure_name END;

Your code and output will look like this:

```
CALL check_rectangle;
```

```
Statement processed.

This is a rectangle.
```

You can use EXEC:

```
EXEC check_rectangle;


Statement processed.

This is a rectangle.
```

Or you can use a BEGIN END block:

```
BEGIN

  check_rectangle;

END;


Statement processed.

This is a rectangle.
```

It's easy to run a procedure after it's been created.

Using Input Parameters with PL/SQL Procedures

We have just created a procedure that checks two values and determines if a shape is a square or rectangle.

What if you wanted to run this procedure with different values?

One way you could do it is:

- Find the code used to create the procedure
- Modify the variables
- Recreate the procedure
- Call the procedure

This is a lot of steps just to see the procedure with different values.

Another way to do this is to use parameters. Parameters let you specify the values used for the variables when you run the procedure, which are then used in the calculation. It's much better to do it this way than to recompile the procedure.

In this example, we'll learn how to provide the length and width values as parameters.

Our code looks like this so far:

```
CREATE OR REPLACE PROCEDURE check_rectangle
AS
  l_width NUMBER(5) := 18;
  l_length NUMBER(5) := 15;
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END check_rectangle;
```

To add parameters, we add brackets after the procedure_name declaration on the first line:

```
CREATE OR REPLACE PROCEDURE check_rectangle ()
AS
  l_width NUMBER(5) := 18;
  l_length NUMBER(5) := 15;
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;

END check_rectangle;
```

Inside the brackets, we add two parameters: one for l_width, and one for l_length. We define these as IN parameters, because they are being provided to the procedure by other code. We also specify them as a NUMBER data type.

```
CREATE OR REPLACE PROCEDURE check_rectangle (

l_width IN NUMBER, l_length IN NUMBER

)

AS

  l_width NUMBER(5) := 18;

  l_length NUMBER(5) := 15;

BEGIN

  IF (l_width < 0 OR l_length < 0) THEN

    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');

  ELSIF (l_width = l_length AND l_length > 50 ) THEN

    DBMS_OUTPUT.PUT_LINE('This is a big square.');

  ELSIF (l_width = l_length) THEN

    DBMS_OUTPUT.PUT_LINE('This is a square.');

  ELSE

    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');

  END IF;

END check_rectangle;
```

I've added the parameters on a separate line to make it more readable, but it's up to you how to do this.

Now we need to remove them from the declaration section. They are already declared in the parameters.

```
CREATE OR REPLACE PROCEDURE check_rectangle (

l_width IN NUMBER, l_length IN NUMBER

)

AS

BEGIN

  IF (l_width < 0 OR l_length < 0) THEN
```

```
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END check_rectangle;
```

Let's run this on the database.

```
Procedure created.
```

The procedure is now created. Let's run this procedure and use parameters. For this example I'll use the CALL keyword but it will work with any method we've seen so far.

Let's try it with the original values of 18 and 15. To use parameters in a call to a procedure, you specify them inside brackets after the procedure name, separated by commas, like this:

```
CALL check_rectangle(18, 15);
```

The output from this statement is:

```
Statement processed.
This is a rectangle.
```

This is called "positional notation", because you're passing parameters in that are getting processed based on their position inside the brackets.

The other way to do this is called "named notation", where you specify the parameter name and the value when you call it:

```
CALL check_rectangle(l_width=>18, l_length=>15);
```

This is done by specifying the parameter name, and then the value, separated by a => symbol to assign the value of the parameter.

This shows the following output:

```
Statement processed.
This is a rectangle.
```

Now you've seen how to run the code with parameters, let's run a few different variations of the procedure to really see how useful parameters are:

```
CALL check_rectangle(18, 15);
CALL check_rectangle(20, 32);
CALL check_rectangle(80, 80);

CALL check_rectangle(-2, 12);
```

The output for this is:

```
Statement processed.

This is a rectangle.

Statement processed.

This is a rectangle.

Statement processed.

This is a big square.

Statement processed.

The width or length is negative.
```

This example shows how useful parameters are. You can simply change the input values to have the procedure run again using these new values.

Using Output Parameters with PL/SQL Procedures

We've learned how to use input parameters to specify inputs to our procedure. Let's learn how to use output parameters.

Why would you use output parameters in PL/SQL code? The main reason is so you can use the output from the procedure elsewhere in your code.

We'll update our procedure to use an output parameter, and then use that in other code.

Here's our procedure at the moment:

```
CREATE OR REPLACE PROCEDURE check_rectangle (

l_width IN NUMBER, l_length IN NUMBER

)

AS

BEGIN

  IF (l_width < 0 OR l_length < 0) THEN

    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');

  ELSIF (l_width = l_length AND l_length > 50 ) THEN

    DBMS_OUTPUT.PUT_LINE('This is a big square.');

  ELSIF (l_width = l_length) THEN
```

```
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END check_rectangle;
```

Let's say we want to return the message to display, instead of displaying it to the screen using PUT_LINE. This is so the code that calls this procedure can decide what to do with it: either display it to the screen, add it to a database table, or something else.

First, we add an OUT parameter for the message. This is so the variable to store this is created, and it can be returned to the code that calls it.

```
CREATE OR REPLACE PROCEDURE check_rectangle (
l_width IN NUMBER, l_length IN NUMBER, l_message OUT VARCHAR2
)
AS
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
    DBMS_OUTPUT.PUT_LINE('The width or length is negative.');
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    DBMS_OUTPUT.PUT_LINE('This is a big square.');
  ELSIF (l_width = l_length) THEN
    DBMS_OUTPUT.PUT_LINE('This is a square.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This is a rectangle.');
  END IF;
END check_rectangle;
```

We can then set this variable to the message we want to show instead of using PUT_LINE:

```
CREATE OR REPLACE PROCEDURE check_rectangle (
l_width IN NUMBER, l_length IN NUMBER, l_message OUT VARCHAR2
)
AS
BEGIN
  IF (l_width < 0 OR l_length < 0) THEN
```

```
    l_message := 'The width or length is negative.';
  ELSIF (l_width = l_length AND l_length > 50 ) THEN
    l_message := 'This is a big square.';
  ELSIF (l_width = l_length) THEN
    l_message := 'This is a square.';
  ELSE
    l_message := 'This is a rectangle.';
  END IF;
END check_rectangle;
```

If we run this code, this is the output we get:

```
Procedure created.
```

We can then output the message using other PL/SQL code, or write it to a database instead:

```
DECLARE
  l_output_message VARCHAR2(100);
BEGIN
  check_rectangle(10, 5, l_output_message);
  DBMS_OUTPUT.PUT_LINE(l_output_message);
END;
```

In this code, we have declared a new variable for the output message. In the BEGIN block, we run the check_rectangle function with our length and width parameters. The third parameter is the l_output_message variable, which is set inside the check_rectangle procedure. We then output this value.

Running this code shows this output:

```
Statement processed.
This is a rectangle.
```

We could insert this into a database table instead if we wanted, because the code is flexible.

```
CREATE TABLE message_output (
  message_val VARCHAR2(100)
);
```

We can then run this code, which inserts into the table instead of displays it on the screen:

```
DECLARE
```

```
  l_output_message VARCHAR2(100);
BEGIN
  check_rectangle(10, 5, l_output_message);
  INSERT INTO message_output(message_val) VALUES (l_output_message);
END;


1 row(s) inserted.
```

If we SELECT from the table, we can see our value.

```
SELECT * FROM message_output;
```

MESSAGE_VAL
This is a rectangle.

So, using output and input parameters can be very useful in PL/SQL procedures as they allow you to do what you need with the values in your code.

How to Create a PL/SQL Function

The other type of code object you can create with PL/SQL is a function.

A PL/SQL function runs some code and returns a value. It can be used in regular SQL as well as PL/SQL.

Creating a function is similar to creating a procedure:

```
CREATE [OR REPLACE] FUNCTION function_name
[ ( parameter_name [ IN | OUT | IN OUT ] parameter_type [, ...] ) ]
RETURN return_datatype
{ IS | AS }
BEGIN
  function_body
END function_name;
```

There are a few things to notice here:

- The function_name is the name of the function to create.
- The OR REPLACE keyword is optional. If you include it, it means that when you run this statement it will replace the function with the same name. If you don't include it and a function with this name already exists, you'll get an error.

One or more parameters can be provided:

- Parameter_name is the name of the parameter, which is used in your function code
- A parameter can be an IN parameter (value is provided to the function to use), an OUT parameter (value is determined by the function and returned to the code that has called it), or IN OUT (combination of IN and OUT).
- A parameter has a parameter_type which is an SQL data type, such as NUMBER.
- Further parameters can be added and separated by commas.

After you define the parameters:

- Add the RETURN keyword and then the data type of the return value. This lets the database know the type of data that will be returned from the function.
- Specify either IS or AS to start the function code.
- BEGIN is specified just like we have learned so far.
- You add the code for your function, and finish with END function_name.

Example PL/SQL Function

Let's write an example PL/SQL function. So far we've been working with code that determines what kind of shape is created when a length and width is provided. Let's write a function that calculates the area of a shape with the length and width provided.

Our function will look like this:

```
CREATE OR REPLACE FUNCTION shape_area

(l_width IN NUMBER, l_length IN NUMBER)

RETURN NUMBER

AS

BEGIN



END;
```

We have the declaration of the function, which includes the name of shape_area. It also includes two parameters, and returns a number.

But the function doesn't do anything.

Let's write some function code.

```
CREATE OR REPLACE FUNCTION shape_area

(l_width IN NUMBER, l_length IN NUMBER)
```

```
RETURN NUMBER
AS
BEGIN
  RETURN l_width * l_length;
END;
```

We've added one line, which calculates the length * width and returns it using the RETURN keyword. This means that any time this function is called, it returns the value of the length multiplied by the width.

Let's create this function. The output shows:

```
Function created.
```

The function is now created. We can call the function from PL/SQL, like this:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(shape_area(5, 4));
END;
```

This code shows the following output:

```
Statement processed.
20
```

Because this code is a function, we can call it from SQL as well:

```
SELECT shape_area(5, 4)
FROM dual;
```

SHAPE_AREA(5,4)
20

Creating a function is just like creating a procedure, except the RETURN statement is needed.

Exceptions in PL/SQL

What is an exception in PL/SQL?

An exception is an error that happens when running your code. Some errors in your code can be found before you run the code, such as missing semicolons or other characters. However, exceptions occur when you run your code and can't be predicted before running the code, such as adding a text value to a number data type.

The good news in PL/SQL is that you can write code to work with these exceptions. This is called "exception handling", and it's where your code can see that an error has happened and take a different action. Without exception handling, your errors are reported to the program that calls it and displayed either in your IDE or in the application.

Syntax for Exception Handling in PL/SQL

Writing code to handle exceptions in PL/SQL looks like this:

```
BEGIN

  executable_code;

EXCEPTION

  WHEN exception_type THEN

    exception_code;

  WHEN OTHERS THEN

    exception_code;

END;
```

The BEGIN section is the same: this is where your executable code goes.

AN EXCEPTION section is added after your executable code, at the end of your code block, before the END statement. This EXCEPTION section contains code for handling any exceptions that occur.

Within the EXCEPTION section is a series of WHEN THEN statements. Each of these statements relates to a possible exception that can be found. When an exception is found when running your code, it's called "throwing an exception". I'm not sure why it's called "throwing", but that's just the programming term for it.

When an exception is found, the WHEN statements are checked. In the WHEN statements you specify the type of exception, like an IF statement. If the type that was found matches the type in your WHEN statement, the code underneath is run.

If the type doesn't match any of your WHEN statements, you can have a WHEN OTHERS THEN statement, which runs in this situation.

Let's take a look at an example.

Example Exception Code in PL/SQL

We'll use a variation of our earlier code that calculates the area of a shape. Instead of using a procedure or a function, we'll just use an anonymous PL/SQL block.

The code looks like this:

```
DECLARE
```

```
  l_length NUMBER := 4;
  l_width NUMBER := 5;
  l_area NUMBER(3);
BEGIN
  l_area := l_length * l_width;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || l_area);
END;
```

Our l_area variable has a maximum size of 3. We can run this code and get this output:

```
Statement processed.
The area is: 20
```

It shows the area is 20, because 5*4 is 20.

Let's say we have a length and width that are both three digits.

```
DECLARE
  l_length NUMBER := 400;
  l_width NUMBER := 500;
  l_area NUMBER(3);
BEGIN
  l_area := l_length * l_width;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || l_area);
END;
```

What happens when we run the code?

We get an error message:

```
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 6 ORA-06512: at "SYS.DBMS_SQL", line 1721
```

This error message has a code: ORA-06502. The message says the number precision is too large. This is because 400 * 500 is 200,000, which is too large for the NUMBER(3) variable.

Rather than display this error message, we can write some code that tells the program what to do. We do this using the EXCEPTION section.

```
DECLARE
  l_length NUMBER := 400;
  l_width NUMBER := 500;
  l_area NUMBER(3);
BEGIN
  l_area := l_length * l_width;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || l_area);
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('There was a problem specifying the area. ' ||
SQLERRM);
END;
```

We have added in the EXCEPTION section. We have also added in the WHEN statement: WHEN VALUE_ERROR. This will pick up any ORA-06502 errors we may experience in our code. The reason that VALUE_ERROR is used instead of the error code is that the name is easy to identify and read.

After the WHEN statement we have a PUT_LINE function call that writes a message, saying there was a problem with the area. We then concatenate something called SQLERRM.

SQLERRM is an SQL function that returns the error message found in the recently-executed code. This will allow us to see the details of the error message.

If we run this code, we get this output:

```
Statement processed.
There was a problem specifying the area. ORA-06502: PL/SQL: numeric or
value error: number precision too large
```

This has been written as an output from PUT_LINE rather than an error message. It means the program will keep running and you can do more with this message, instead of just stopping the program.

Raising Exceptions in PL/SQL

We've seen what happens if we perform a calculation to set a value that's higher than the maximum size, or do something that Oracle does not expect. This uses a built-in exception.

What if we find something that our business rules say is not valid, but it's OK with the Oracle database?

One example of this is what if one of the parameters is negative?

Our code could look like this:

```
DECLARE
  l_length NUMBER := -4;
  l_width NUMBER := 3;
  l_area NUMBER(3);
BEGIN
  l_area := l_length * l_width;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || l_area);
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('There was a problem specifying the area. ' ||
SQLERRM);
END;
```

The output is:

```
Statement processed.
The area is: -12
```

This shows the area as -12, which is not a valid value. We don't want to have negative area values.

We can avoid this by creating or raising our own exception. To do this:

- Declare a new variable which has a type of EXCEPTION
- Check the criteria that cause this issue, and raise the exception if they are true

To do this in our code, we declare a new variable with a type of EXCEPTION.

```
DECLARE
  l_length NUMBER := -4;
  l_width NUMBER := 3;
  l_area NUMBER(3);
  ex_negative_area EXCEPTION;
BEGIN
  l_area := l_length * l_width;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || l_area);
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('There was a problem specifying the area. ' ||
SQLERRM);
END;
```

Now, we've created a new variable called ex_negative_area. This is an EXCEPTION. We now need to check if the area is negative, and if it is, raise the exception.

```
DECLARE
  l_length NUMBER := -4;
  l_width NUMBER := 3;
  l_area NUMBER(3);
  ex_negative_area EXCEPTION;
BEGIN
  l_area := l_length * l_width;
  IF (l_area < 0) THEN
    RAISE ex_negative_area;
  END IF;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || l_area);
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('There was a problem specifying the area. ' ||
SQLERRM);
END;
```

This code checks if the area is less than zero. If it is, then the exception is raised using the RAISE ex_negative_area statement.

Finally, we need to add this to our EXCEPTION section. If we just raise the exception without writing a WHEN statement, this is what we will see:

```
User-Defined Exception ORA-06512: at line 9 ORA-06512: at
"SYS.DBMS_SQL", line 1721
```

The user-defined exception error message means we raised the exception but did nothing with it.

Our new code will look like this:

```
DECLARE
  l_length NUMBER := -4;
  l_width NUMBER := 3;
  l_area NUMBER(3);
  ex_negative_area EXCEPTION;
BEGIN
  l_area := l_length * l_width;
  IF (l_area < 0) THEN
```

```
    RAISE ex_negative_area;
  END IF;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || l_area);
EXCEPTION
  WHEN ex_negative_area THEN
    DBMS_OUTPUT.PUT_LINE('The area is negative. Please confirm the input
values are positive.');
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('There was a problem specifying the area. ' ||
SQLERRM);
END;
```

We can now run this code.

```
Statement processed.
The area is negative. Please confirm the input values are positive.
```

The message shown is what we have entered in the EXCEPTION section. It says the area is negative, which is what we wanted to display.

Conclusion

Procedures and functions in PL/SQL are objects that contain PL/SQL code for you to run at a later stage. They have several advantages such as the ability to reuse code and to simplify your programming. There are some differences between procedures and functions.

Exceptions are errors that are encountered as your program is running. They can be handled within your PL/SQL code, otherwise they will display an error in your IDE or in your application.

Quiz

Question 1

Which one of these statements is true about the differences between a procedure and a function?

- They are both the same.
- Functions must have a RETURN value, but procedures don't need one.
- Procedures can only take IN parameters and not OUT parameters.
- Procedures can be used in regular SQL, but functions cannot.

Question 2

What does the OR REPLACE keyword do as part of the CREATE PROCEDURE or CREATE FUNCTION statement?

- It replaces the existing object with the one you have defined in the statement if it already exists.
- Nothing, this keyword is not valid.
- Nothing, it's a valid keyword but does not apply to procedures or functions.
- It determines if your code will insert new data into a table or update existing data.

Question 3

What is an exception?

- An error that happens when you create your procedure or function, such as a missing bracket.
- The part of an IF statement that runs if no criteria are met.
- An error that happens while your code is running.
- A collection of data that is not inserted into a table.

Question 4

How can you run the code in a procedure?

- Run the "CALL procedure_name" statement.
- It's run automatically when you create the procedure.
- Run the "RUN procedure_name" statement.
- You can't run procedures, you can only run functions.

Chapter 5: Cursors, Arrays, and Bulk Collect

In this chapter, we'll learn about some pretty useful concepts in PL/SQL:

- What a cursor is
- The two types of cursors
- Inserting, updating, and deleting data in tables using PL/SQL
- Selecting data from a table into PL/SQL
- What an array is and how to use one

What Is a Cursor in PL/SQL?

A cursor is an area in memory that Oracle creates when an SQL statement is run. The cursor contains several pieces of information including the rows that the statement has returned (if it's a SELECT statement), and attributes about the result set such as the number of rows impacted by the statement.

There are two types of cursors in in Oracle SQL: implicit cursors and explicit cursors.

Implicit Cursors

An implicit cursor is a type of cursor automatically created by Oracle when an SQL statement is run. It's called an implicit cursor because it's created automatically - you don't need to do anything for it to get created.

Whenever your run a DML statement (INSERT, UPDATE, DELETE, or SELECT), an implicit cursor is created:

- For INSERT statements, the cursor contains the data that is being inserted.
- For UPDATE or DELETE statements, the cursor identifies the rows that will be affected
- For SELECT statements, the rows that are retrieved can be stored in an object.

How can implicit cursors help us? For INSERT, UPDATE, and DELETE statements, we can see several attributes of the statements to understand if they have worked or not.

Let's see an example of this.

Example - Using a Cursor with INSERT

Let's see what can be done with an implicit cursor and an INSERT statement.

First, we'll need to create a table:

```
CREATE TABLE person (
fname VARCHAR2(50)

);
```

If you run this in an SQL window, the table will be created.

Now let's write some PL/SQL to insert data into this table.

```
BEGIN

  INSERT INTO person (fname)

  VALUES ('John');

  DBMS_OUTPUT.PUT_LINE('Total number of rows impacted: ' ||
sql%rowcount);

END;
```

This code includes the term sql%rowcount. The "sql" is the object name of the SQL statement that ran last. The "rowcount" is the attribute of this most recent SQL statement, and contains the number of rows impacted by the statement. The % sign indicates that the rowcount is an attribute of the sql object. So, sql%rowcount will contain the number of rows impacted by the most recent SQL statement.

If we run this code, we should see this output:

```
1 row(s) inserted.

Total number of rows impacted: 1
```

The output shows the number of impacted rows. We can do more with this value, such as storing it in a variable:

```
DECLARE

  l_total_rows NUMBER(10);

BEGIN

  INSERT INTO person (fname)

  VALUES ('John');

  l_total_rows := sql%rowcount;

  DBMS_OUTPUT.PUT_LINE('Total number of rows impacted: ' ||
l_total_rows);

END;
```

You can also use this information in an IF statement and run different code based on the value.

```
DECLARE

  l_total_rows NUMBER(10);

BEGIN

  INSERT INTO person (fname)
  VALUES ('John');
  l_total_rows := sql%rowcount;
  IF (l_total_rows > 0) THEN
    DBMS_OUTPUT.PUT_LINE('Rows inserted: ' || l_total_rows);
  ELSE
    DBMS_OUTPUT.PUT_LINE('No rows inserted.');
  END IF;
END;
```

If you run a SELECT statement on your table, you should see the data in the table. Assuming the statement was only run once, this is what you'll see:

FNAME
John

Example - Using a Cursor with UPDATE

We can make use of implicit cursors in PL/SQL with UPDATE statements. For example, this code shows us the number of rows impacted by the UPDATE statement.

```
BEGIN
  UPDATE person
  SET fname='Susan'
  WHERE fname = 'John';
  DBMS_OUTPUT.PUT_LINE('Total number of rows impacted: ' ||
sql%rowcount);
END;
```

If we run this code, we'll get this output:

```
1 row(s) updated.
Total number of rows impacted: 1
```

It says that 1 row is impacted by our update statement. The row is also updated: that UPDATE statement is executed. If we run a SELECT query on the table, we'll see the new value:

FNAME
Susan

Now, let's run our earlier code and see what happens:

```
1 row(s) updated.

Total number of rows impacted: 0
```

It says 0 rows are impacted (even though the output also says "1 row(s) updated"). This is because the WHERE clause means the statement is looking for a row of "John", but the value in the row is equal to "Susan", so it is not updated.

Another way of checking for rows being updated is to use the %FOUND and %NOTFOUND attributes, rather than %ROWCOUNT.

The %FOUND attribute will return true if at least one row is impacted by an INSERT, UPDATE, or DELETE statement, otherwise it will return false. The %NOTFOUND attribute is the opposite: it will return true if no rows are impacted, and false if at least one row is impacted.

We can change our code to use the %FOUND attribute:

```
BEGIN
  UPDATE person
  SET fname='Susan'
  WHERE fname = 'John';
  IF (sql%found) THEN
    DBMS_OUTPUT.PUT_LINE('Rows updated: ' || sql%rowcount);
  ELSE
    DBMS_OUTPUT.PUT_LINE('Rows not updated.');
  END IF;
END;
```

This output is shown:

```
1 row(s) updated.

Rows not updated.
```

The output message says no rows are updated. This is because the WHERE clause doesn't match any of the existing rows.

We can change our UPDATE statement so rows are found:

```
BEGIN
  UPDATE person
  SET fname='Mark'
  WHERE fname = 'Susan';
  IF (sql%found) THEN
    DBMS_OUTPUT.PUT_LINE('Rows updated: ' || sql%rowcount);
  ELSE
    DBMS_OUTPUT.PUT_LINE('Rows not updated.');
  END IF;
END;
```

```
1 row(s) updated.
Rows updated: 1
```

Example - Using a Cursor with DELETE

The same concept of an implicit cursor applies to a DELETE statement. We can write some code that deletes a row and then displays an output depending on if a row was deleted or not.

```
BEGIN
  DELETE FROM person
  WHERE fname = 'Susan';
  IF (sql%found) THEN
    DBMS_OUTPUT.PUT_LINE('Rows updated: ' || sql%rowcount);
  ELSE
    DBMS_OUTPUT.PUT_LINE('Rows not updated.');
  END IF;
END;
```

This output shows that no rows were deleted.

```
1 row(s) deleted.
Rows not updated.
```

This is because no rows have an fname value of "Susan". If we change it to "Mark" and run the statement again, the row will be deleted.

```
BEGIN
  DELETE FROM person
  WHERE fname = 'Mark';
  IF (sql%found) THEN
```

```
    DBMS_OUTPUT.PUT_LINE('Rows updated: ' || sql%rowcount);
  ELSE
    DBMS_OUTPUT.PUT_LINE('Rows not updated.');
  END IF;
END;
```

```
1 row(s) deleted.
Rows updated: 1.
```

Selecting Data with an Implicit Cursor

What about a SELECT query with an implicit cursor? That works a little differently.

We can use a SELECT query with an implicit cursor. However, we need to use a variable to store the returned value. We can use SELECT INTO for this.

First, we declare a variable to hold our data. Then, we select a column value from the table and store it in this variable. In this example, we'll just select a single value, as we need to treat multiple rows differently.

Ensure your person table only has one row in it:

```
DELETE FROM person;
INSERT INTO person (fname) VALUES ('John');
SELECT * FROM person;
```

Now, let's write some PL/SQL code to select this value into a variable.

```
DECLARE
  l_fname VARCHAR2(50);
BEGIN
  SELECT fname
  INTO l_fname
  FROM person;
  DBMS_OUTPUT.PUT_LINE('The name is ' || l_fname);
END;
```

First, we've declared a variable called l_fname. Then in the BEGIN section, we have selected the fname column from the person table into the l_fname variable. This variable now holds the value from that query. We then output it using the PUT_LINE function.

The output from this statement is:

```
Statement processed.

The name is John
```

We can see the output says "The name is John".

What if there are two or more values in the table? How do we handle this? Our variable can only hold one value.

Here's the code to insert another row into the table:

```
INSERT INTO person (fname) VALUES ('Susan');

SELECT * FROM person;
```

FNAME
John
Susan

Now, let's run our PL/SQL code to SELECT the value from this column into a variable.

```
DECLARE

  l_fname VARCHAR2(50);

BEGIN

  SELECT fname

  INTO l_fname

  FROM person;

  DBMS_OUTPUT.PUT_LINE('The name is ' || l_fname);

END;
```

This is our output. The error message we get is:

```
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4 ORA-06512: at "SYS.DBMS_SQL", line 1721
```

This error happens because we have two values returned from our query (John and Susan), but only one variable to store them.

How do we handle situations like this? We'll learn about that later in this guide.

For now, let's look at explicit cursors.

Explicit Cursors

In Oracle PL/SQL, an explicit cursor is one that is declared by the PL/SQL code. You have more control over it and how the data is handled.

Using an explicit cursor involves several steps:

1. Declaring the cursor as a variable in the DECLARE section.
2. Opening the cursor, which allocates memory for it.
3. Fetching the cursor, which means the SELECT statement is run and data is stored in the cursor.
4. Closing the cursor, which releases the allocated memory.

Example: Explicit Cursors in PL/SQL

To declare a cursor, you add a line into the DECLARE section of your code. It includes your SELECT statement.

```
DECLARE
  CURSOR c_person IS
  SELECT fname FROM person;
BEGIN


END;
```

As shown above, cursors are declared using this syntax:

```
CURSOR cursor_name IS select_statement;
```

Next, we need to open the cursor. This is done in the BEGIN block:

```
DECLARE
  CURSOR c_person IS
  SELECT fname FROM person;
BEGIN
  OPEN c_person;
END;
```

Now, we fetch the data from the cursor into a variable.

```
DECLARE
  l_fname VARCHAR2(50);
  CURSOR c_person IS
  SELECT fname FROM person;
```

```
BEGIN
  OPEN c_person;
  FETCH c_person INTO l_fname;
END;
```

We then have to close the cursor.

```
DECLARE
  l_fname VARCHAR2(50);
  CURSOR c_person IS
  SELECT fname FROM person;
BEGIN
  OPEN c_person;
  FETCH c_person INTO l_fname;
  CLOSE c_person;
END;
```

Let's write the variable to the screen.

```
DECLARE
  l_fname VARCHAR2(50);
  CURSOR c_person IS
  SELECT fname FROM person;
BEGIN
  OPEN c_person;
  FETCH c_person INTO l_fname;
  DBMS_OUTPUT.PUT_LINE('Name is: ' || l_fname);
  CLOSE c_person;
END;
```

The output shown here is:

```
Statement processed.
Name is: John
```

It shows the name of John.

But there are two records in the table. How can we handle this with our cursor code?

Example: Explicit Cursors and Loops

If our SELECT query returns multiple rows from an explicit cursor, we need to use a loop to get all of the data.

We use the same loop as we learned earlier in this guide:

```
LOOP
  FETCH cursor_name INTO variables;
  EXIT WHEN cursor_name%notfound;
  your_code;
END LOOP;
```

What does this code do?

- It starts with a LOOP, like we've learned in an earlier section.
- We then have a FETCH statement, which will fetch the values of the columns of your SELECT statement from a single row into the variables you specify
- We then have EXIT WHEN cursor_name%notfound, which means the loop will exit once there are no more rows found in the cursor.
- The code you want to execute is then run
- The END LOOP statement will end the loop.

The important part about this code is that the FETCH statement will fetch the next row from the cursor. This means it fetches one row at a time.

Let's see an example.

```
DECLARE
  l_fname VARCHAR2(50);
  CURSOR c_person IS
  SELECT fname FROM person;
BEGIN
  OPEN c_person;
  LOOP
    FETCH c_person INTO l_fname;
    EXIT WHEN c_person%notfound;
    DBMS_OUTPUT.PUT_LINE('Name is: ' || l_fname);
  END LOOP;
  CLOSE c_person;
END;
```

This is similar to the earlier code:

- We declare the l_fname variable, and then the cursor as the SELECT statement.
- We open the cursor
- We start the loop
- Inside the loop, we fetch the cursor into the l_fname variable
- We output the value of l_fname
- We loop until no more records are found
- We close the cursor and end the program

This might seem like a lot of code just to write out some values to the screen, and it is. It's simpler to just do this as a SELECT statement. But this is just to demonstrate the concepts. You can use any code in your PL/SQL program instead of writing the values to screen.

Here's what is shown when that code is run:

```
Statement processed.
Name is: John
Name is: Susan
```

It shows two rows of output as there are two rows in the table.

So that's how you can use explicit cursors with SELECT statements and use loops to fetch data from tables that have multiple rows.

Arrays: Multiple Values in a Single Variable

In our earlier examples, we have used a cursor and run a SELECT statement to fetch a value into a variable.

What if we want to store multiple values but don't want to have to create separate variables for each of the values? We want to avoid doing this:

```
DECLARE
  l_fname_1 VARCHAR2(50);
  l_fname_2 VARCHAR2(50);
  l_fname_3 VARCHAR2(50);
..
```

That will end up with a lot of unnecessary code.

There is a better way: using arrays.

Arrays are a concept in programming where a single variable stores multiple values. Each of the values are accessed using an index number. The concept works like this:

- Declare an array, such as array_fname.
- Populate the array, so each value in the array has an index number:
    - array_fname(1) = 'John',
    - array_fname(2) = 'Susan'
    - array_fname(3) = 'Mark'
- Access the values throughout your code.

So how can you create an array in PL/SQL? Arrays are created like this:

```
DECLARE
  TYPE fname_array_type IS VARRAY(10) OF VARCHAR2(50);
  fname_array fname_array_type;
BEGIN
  your_code;
END;
```

The code includes two lines to set up your array: creating a type, and creating the variable.

Example: Creating a VARRAY

In PL/SQL, to use an array, you need to declare the type of array first. This is done using the TYPE statement. In other programming languages, the array declaration and type are done on a single line, but in PL/SQL they are on separate lines.

This line is:

```
TYPE fname_array_type IS VARRAY(10) OF VARCHAR2(50);
```

It says that you are creating a TYPE, and the name of this type we have called fname_array_type. This is just a name we made up, but it's clear it's a type. I could have called it varchar_type or something else to indicate what it is.

We then specify IS VARRAY(10). This means the type is a VARRAY, which is the array type in PL/SQL. The 10 indicates the maximum number of elements in this array. This means any variable based on this type can hold up to 10 values in it.

Finally, we specify OF VARCHAR2(50) which is the type of data it holds. You can create types that hold NUMBER values or any other PL/SQL data type.

Once we have declared our type, we can then declare a variable of that type:

```
fname_array fname_array_type;
```

This code declares a new variable called fname_array with the data type of fname_array_type. We don't need to specify anything else because that's included in the TYPE declaration.

Now what? We populate the array.

```
DECLARE

  TYPE fname_array_type IS VARRAY(10) OF VARCHAR2(50);

  fname_array fname_array_type;

BEGIN

  fname_array := fname_array_type('Adam', 'Belinda', 'Charles',
'Debra');

END;
```

We've set the fname_array to contain four different names.

What can we do with these values now? We can use a FOR loop to loop through them and write them to the screen. We can do a lot more with these values, but for this example, we'll just write them to the screen to keep it simple.

```
DECLARE

  TYPE fname_array_type IS VARRAY(10) OF VARCHAR2(50);

  fname_array fname_array_type;

BEGIN

  fname_array := fname_array_type('Adam', 'Belinda', 'Charles',
'Debra');

  FOR i IN 1 .. fname_array.count LOOP

    DBMS_OUTPUT.PUT_LINE('Name is: ' || fname_array(i));

  END LOOP;

END;
```

The code we have added is the FOR loop. As we learned in an earlier section, the FOR loop starts at a number and ends at another number. In this example, the variable is called "i", and it starts at 1 and ends at something called fname_array.count.

This count is an attribute of the array, and it returns the number of records or elements in the array. It's better to use this rather than just the number 4, because the array size can change (if you're populating it from somewhere else).

Inside the loop, we show the value by using fname_array(i), which means we are referring to a single element inside the array. We are referring to the element with an index position of whatever "i" is equal to, which changes each time the loop is run.

The output should show separate lines that display each of the values in the array.

If we run the code, this is what we get:

```
Statement processed.

Name is: Adam

Name is: Belinda

Name is: Charles

Name is: Debra
```

It shows separate lines, one for each name in the array.

So that's how you can use arrays in your PL/SQL code.

Conclusion

A cursor is an area in memory used to hold details of an SQL statement, such as the number of impacted rows and the values returned from a SELECT query.

Oracle offers two types of cursors: an implicit and an explicit cursor. Implicit cursors are created automatically without you doing anything. Explicit cursors are specified by you but you have more control over them.

Arrays are types of variables that allow you to store many values in a single variable. They are declared as a TYPE first, and then the array variable is declared. They can be accessed using a number that represents their index, and are often done using loops.

Quiz

Question 1

How can you declare an explicit cursor?

- cursor_name IS select_query;
- cursor_name := select_query;
- cursor_name IS TYPE OF explicit_cursor;
- You can't, because explicit cursors are created automatically by Oracle.

Question 2

What does the %FOUND attribute of a cursor do?

- Returns the data from the cursor.
- Returns the number of rows found by the cursor
- Nothing, it won't work because it needs to be written in lowercase.
- Returns true if at least one row is impacted by the query, otherwise it shows false

Question 3

What does this code do?

```
TYPE id_arraytype IS VARRAY(10) OF NUMBER(4);
id_array id_arraytype;
```

- Nothing, it will show an error
- It will declare a new type of an array of numbers, and a new variable of that type
- It declares a new type of array, but does not declare a variable of that type
- It creates a new table with two columns, one as an array and the other as a number.

Question 4

How can you get the number of rows impacted by an INSERT statement in your PL/SQL code?

- Use the sql%ROWCOUNT attribute.
- SELECT from the table to find rows inserted today.
- You can't.
- Use the sql%NUMROWS attribute.

Chapter 6: Record Types and Bulk Collect

In this chapter, we'll learn about:

- Using data types based on tables and columns
- Understanding what collections are
- Using the BULK COLLECT keyword to fetch data in bulk

These are all great features of the PL/SQL language and will take your code to the next level.

Field Types

In earlier code, we declared variables to hold values we loaded from the database.

```
DECLARE
  l_fname VARCHAR2(50);
BEGIN
  SELECT fname
  INTO l_fname
  FROM person;
  DBMS_OUTPUT.PUT_LINE('The name is ' || l_fname);
END;
```

The l_fname variable is a VARCHAR2(50). This just happens to be the same as the table we created:

```
CREATE TABLE person (
  fname VARCHAR2(50)
);
```

But what if the table declaration changes? We shouldn't be forcing our code to be the same as the table in this way, because we would have to change our code.

There is a better way to do it, and that's using the %TYPE attribute.

The TYPE attribute will let you declare a variable based on the type of a column in a table. This means it is automatically linked to, or set the same as a column that exist. There's no need to look up the data type manually or change the code if your table changes.

The code to do that looks like this:

```
DECLARE
```

```
  variable_name table_name.column_name %TYPE;

BEGIN
```

...

After specifying the variable name, you specify the column and table name separated by a period. You then specify %TYPE, which indicates that your variable has the same type as this column.

Example - Using the %TYPE Attribute

We can update our code to use this:

```
DECLARE

  l_fname person.fname%TYPE;

BEGIN

  SELECT fname

  INTO l_fname

  FROM person;

  DBMS_OUTPUT.PUT_LINE('The name is ' || l_fname);

END;
```

The output of this code is:

```
Statement processed.

The name is John
```

This makes your code easier to maintain.

Example - Two Variables Using %TYPE

You can declare multiple variables that use the %TYPE attribute if you need to.

Let's say our table looked like this:

```
CREATE TABLE person (

fname VARCHAR2(50),

lname VARCHAR2(50),

salary NUMBER(8)

);

INSERT INTO person (fname, lname, salary) VALUES ('John', 'Smith',
50000);
```

We have one record:

FNAME	LNAME	SALARY
John	Smith	50000

Our PL/SQL code can output this record by using variables and the %TYPE attribute.

```
DECLARE
  l_fname person.fname%TYPE;
  l_lname person.lname%TYPE;
  l_salary person.salary%TYPE;
BEGIN
  SELECT fname, lname, salary
  INTO l_fname, l_lname, l_salary
  FROM person;
  DBMS_OUTPUT.PUT_LINE('The name is ' ||
    l_fname || ' ' || l_lname || ' with a salary of ' || l_salary);
END;
```

We have used three variables (one for each column), each related to the column in the table.

The output of this code is:

```
Statement processed.
The name is John Smith with a salary of 50000
```

Row Types

Another useful feature of PL/SQL is the ability to create a variable that has the same data type of an entire table row.

In the example above, we had three separate variables for each of the different columns:

```
DECLARE
  l_fname person.fname%TYPE;
  l_lname person.lname%TYPE;
  l_salary person.salary%TYPE;
BEGIN
```

...

This may seem OK, but what if we want to use 5 or 10 columns? We would need to declare a lot of variables.

In PL/SQL, you can declare a single variable and set the type of it equal to the entire row. It kind of works like an array, where each element or attribute is equal to one column.

We can do this using the %ROWTYPE attribute:

```
DECLARE

  variable_name table_name%ROWTYPE;

BEGIN
```

...

We don't need to specify any column names here: just the variable name, the table to base it on, and %ROWTYPE.

How do we access the values? We use variable_name.column_name.

Let's see an example:

```
DECLARE

  l_row_person person%ROWTYPE;

BEGIN

  SELECT fname, lname, salary

  INTO l_row_person.fname, l_row_person.lname, l_row_person.salary

  FROM person;

  DBMS_OUTPUT.PUT_LINE('The name is ' ||

    l_row_person.fname || ' ' || l_row_person.lname || ' with a salary 
of ' || l_row_person.salary);

END;
```

This code includes a few changes:

- There is a single variable called l_row_person, which has a type equal to the person table (using %ROWTYPE).
- The SELECT INTO is selecting the column values into separate attributes of the l_row_person variable, one for each column.
- The PUT_LINE function uses the attributes of the l_row_person variable. For example, l_row_person.fname is the fname column from the person table.

This results in less code and seems easier to read. It can make a big difference in larger programs.

The output of this code is:

```
Statement processed.

The name is John Smith with a salary of 50000
```

Cursor Based Records

In the earlier example, we used %ROWTYPE to create a variable based on the data type of a table's columns.

In an earlier chapter, we also looked at using cursors to select records. Here's an example:

```
DECLARE
  l_fname VARCHAR2(50);
  CURSOR c_person IS
  SELECT fname FROM person;
BEGIN
  OPEN c_person;
  FETCH c_person INTO l_fname;
  DBMS_OUTPUT.PUT_LINE('Name is: ' || l_fname);
  CLOSE c_person;
END;
```

This code will select the fname into a variable. Another way to do this is to declare a variable that is equal to the type of the cursor, rather than the table or the column. This is helpful if you have a complicated query that gets data from multiple tables.

The code would look like this:

```
DECLARE
  CURSOR c_person IS
  SELECT fname, lname, salary FROM person;
  c_person_rec c_person%ROWTYPE;
BEGIN
  OPEN c_person;
  FETCH c_person INTO c_person_rec;
  DBMS_OUTPUT.PUT_LINE('The name is ' ||
    c_person_rec.fname || ' ' || c_person_rec.lname || ' with a salary
of ' || c_person_rec.salary);
  CLOSE c_person;
```

```
END;
```

This works similar to using a table. The output of this query is:

```
Statement processed.

The name is John Smith with a salary of 50000
```

PL/SQL Collections

PL/SQL includes a feature called collections. A collection in PL/SQL is a set of values that have the same data type. It's similar to an array that we've learned about earlier in this guide, but there are several differences.

Object Type	Number of Elements	Index Type	Dense or Sparse	Can be an Object
Varray	Fixed	Number	Dense	Yes
Index By Table	Variable	String	Either	No
Nested Table	Variable	Number	Either	Yes

We'll learn about the two types of collections in this section: an Index By Table and a Nested Table.

Index By Table

An Index By Table (also known as an associative array) is a type of variable that stores key-value pairs. Like arrays in other programming languages, the keys can be numbers or strings.

Creating a variable using an Index By Table is done in the following way:

```
TYPE type_name IS TABLE OF element_data_type [NOT NULL] INDEX BY
index_data_type;

variable_name type_name;
```

In this code, we declare a type, and then a variable of that type. The type has a name that we can provide, and we specify the data type of the elements and the data type of the index.

An example of this feature is:

```
DECLARE

  TYPE first_name IS TABLE OF VARCHAR2(50) INDEX BY PLS_INTEGER;

  name_list first_name;

BEGIN
```

```
..
END;
```

How can we populate and use this list? We add items, or elements, to the list by specifying the key in brackets and using the := symbol:

```
name_list(1) := 'John';
```

We can add several elements to this variable:

```
DECLARE
  TYPE first_name IS TABLE OF VARCHAR2(50) INDEX BY PLS_INTEGER;
  name_list first_name;
BEGIN
  name_list(1) := 'John';
  name_list(2) := 'Susan';
  name_list(3) := 'Mark';
  name_list(4) := 'Debra';
END;
```

How do we access the values in the list to output them? We can use the FIRST and NEXT functions of the variable, as well as a loop. These functions are built into the Index By Table data type.

Our code looks like this:

```
DECLARE
  TYPE first_name IS TABLE OF VARCHAR2(50) INDEX BY PLS_INTEGER;
  name_list first_name;
  current_name_id PLS_INTEGER;
BEGIN
  name_list(1) := 'John';
  name_list(2) := 'Susan';
  name_list(3) := 'Mark';
  name_list(4) := 'Debra';
  current_name_id := name_list.FIRST;
  WHILE current_name_id IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE('The name is ' || name_list(current_name_id));
    current_name_id := name_list.NEXT(current_name_id);
  END LOOP;


END;
```

There is a lot to take in here:

- We have declared a new variable called current_name_id which stores the index of the collection, and is used for the loop.
- The name_list.FIRST will return the index of the first element, which has been stored in the current_name_id variable;
- A WHILE loop will loop through the name_list and output the value of the name_list element.
- The current_name_id variable is incremented using the name_list.NEXT function.

The output of this code is:

```
Statement processed.

The name is John

The name is Susan

The name is Mark

The name is Debra
```

Example - Index By Table with Select Query

You can also use an Index By Table with a SELECT query on the database.

Using our person table from earlier, this code will select the data from that table into the collection:

```
DECLARE

  CURSOR c_person IS

  SELECT fname, lname, salary FROM person;

  TYPE col_person IS TABLE OF person.fname%TYPE INDEX BY PLS_INTEGER;

  person_list col_person;

  rowcounter PLS_INTEGER := 0;

BEGIN

  FOR i IN c_person LOOP

    rowcounter := rowcounter + 1;

    person_list(rowcounter) := i.fname;

    DBMS_OUTPUT.PUT_LINE('The name is ' || person_list(rowcounter));
  END LOOP;
END;
```

The output will show:

```
Statement processed.
The name is John
The name is Susan
The name is Mark
The name is Debra
```

So that's how you can use an index by table, or associative array, in PL/SQL.

Nested Table

A nested table in PL/SQL is another type of collection. It's very similar to an Index By Table, except it always has an integer for an index. It does not have an INDEX BY clause;

The syntax for creating one is:

```
TYPE type_name IS TABLE OF element_data_type [NOT NULL];
variable_name type_name;
```

We declare the name of the type and the element type, then we declare a variable of that type.

An example of this in action can be done by modifying the example from earlier:

```
DECLARE
  TYPE first_name IS TABLE OF VARCHAR2(50);
  name_list first_name;
BEGIN
  name_list := first_name('John', 'Susan', 'Mark', 'Debra');
  FOR i IN 1 .. name_list.count LOOP
    DBMS_OUTPUT.PUT_LINE('The name is ' || name_list(i));
  END LOOP;
END;
```

This is less code than earlier examples, but it shows the concept of a Nested Table.

The output of this code is:

```
Statement processed.
The name is John
The name is Susan

The name is Mark

The name is Debra
```

When To Use Index By Tables, Nested Tables, or VArrays in PL/SQL

We've learned about Index By Tables, Nested Tables, and Varrays. How do we know when to use each of them?

Oracle has some recommendations.

When to use an Index By Table/Associative Array:

- When you have a small lookup table, as it's created each time in memory whenever you run your code

When to use a Nested Table:

- When the index values are not consecutive
- When there is not a set number of index values
- You need to delete some of the elements

When to use a Varray:

- The number of elements is known in advance
- The elements are usually accessed in order

Bulk Collect

Our PL/SQL code often contains PL/SQL code (declaring variables, loops, IF statements) and SQL code (SELECT, INSERT, UPDATE). This makes our programs quite powerful.

However, it can also make our programs quite slow if they are not written correctly.

Each time an SQL statement is run from PL/SQL, a "context switch" is performed. The server switches from running the PL/SQL code to running the SQL code. This involves a small amount of work by the server. This may not be noticeable with one statement, but if you're running hundreds or thousands of statements, across many users, then it can really add up.

Let's say we had this PL/SQL code that updated the salary in our person table.

Here's our setup data.

```
DELETE FROM person;

INSERT INTO person (fname, lname, salary) VALUES ('John', 'Smith',
20000);
```

```
INSERT INTO person (fname, lname, salary) VALUES ('Susan', 'Jones',
30000);

INSERT INTO person (fname, lname, salary) VALUES ('Mark', 'Blake',
25000);

INSERT INTO person (fname, lname, salary) VALUES ('Debra', 'Carlson',
40000);
```

Here's our PL/SQL code.

```
BEGIN
  FOR current_rec IN (SELECT fname
  FROM person) LOOP

    DBMS_OUTPUT.PUT_LINE('Name: ' || current_rec.fname);

    UPDATE person p
    SET p.salary = p.salary + 1000
    WHERE p.fname = current_rec.fname;

  END LOOP;
END;
```

The output is:

```
1 row(s) updated.
Name: John
Name: Susan
Name: Mark
Name: Debra
```

In this code, we have a query that selects names. We loop through this collection and update the person table for each entry in the collection, which means it runs the UPDATE statement four times.

We can run a SELECT statement to check the new values.

```
SELECT * FROM person;
```

FNAME	LNAME	SALARY
John	Smith	21000
Susan	Jones	31000

Mark	Blake	26000
Debra	Carlson	41000

We can see that each of the salary values have been increased by 1. However the UPDATE statement has ran 4 separate times. This is something we want to avoid.

We also want to avoid using a cursor for loop as we have here. A cursor for loop is where we have a FOR loop that includes a SELECT query in its criteria. It allows you to loop through each record and process it, as we have done in our code.

However, a cursor for loop is slow as it processes each row individually. It also performs a context switch between the SQL engine (running the SELECT query) and the PL/SQL engine (the FOR loop) There are better ways to do this: using a feature called BULK COLLECT.

BULK COLLECT allows you to fetch all rows from a SELECT query into a collection in a single statement. The SQL code is run once, and the remainder of the logic is performed in PL/SQL. This greatly reduces the context switching between SQL and PL/SQL, improving the performance of your code. Steve Feuerstein (and many others) have written about the benefits of using BULK COLLECT.

Example - BULK COLLECT

So how do we do this? We can introduce a variable to hold our data, and replace the cursor for loop with a SELECT and BULK COLLECT:

```
DECLARE
  TYPE name_type IS TABLE OF person.fname%TYPE;
  name_list name_type;
BEGIN
  SELECT fname
  BULK COLLECT INTO name_list
  FROM person;
  FOR i IN 1 .. name_list.count LOOP

    UPDATE person p
    SET p.salary = p.salary + 1000
    WHERE p.fname = name_list(i);

  END LOOP;
```

```
END;
```

The changes we have made here are:

- We've declared a new TYPE called name_type, which is a table of values of the same type as the person.fname column.
- We've declared a new variable based on that type, called name_list.
- We've run a SELECT statement which selects the fname column into the name_list variable, using BULK COLLECT. This means all values from the query are added into this collection.

The loop is then run based on the data already loaded into the name_list variable.

The output is:

```
1 row(s) updated.

Name: John

Name: Susan

Name: Mark

Name: Debra
```

This improves the performance of the SELECT statement. But what about the UPDATE statement?

FORALL in PL/SQL

The UPDATE statement in the earlier code is run each time during the loop. This means that there is a context switch between PL/SQL and SQL each time the statement is run. In this example it happens 4 times, but in a real database it can happen hundreds or thousands of times.

If you can put your SQL code outside the loop, then that is better. The code will only run once on all the rows that need to be updated.

However, if you have logic that means you can't do this, you can use a feature called FORALL.

In PL/SQL, the FORALL statement will tell the database to generate and run all of the DML statements that would have been generated separately and run them at once. This reduces the time taken and the context switches.

To do this, change the code to ensure the UPDATE statement is inside a FORALL:

```
DECLARE

  TYPE name_type IS TABLE OF person.fname%TYPE;

  name_list name_type;

BEGIN

  SELECT fname

  BULK COLLECT INTO name_list
  FROM person;
  FORALL i IN 1 .. name_list.count
    UPDATE person p
    SET p.salary = p.salary + 1000
    WHERE p.fname = name_list(i);
END;
```

This will run the UPDATE statements once on the database.

You can check the before and after using this code:

```
SELECT * FROM person;

DECLARE
  TYPE name_type IS TABLE OF person.fname%TYPE;
  name_list name_type;
BEGIN
  SELECT fname
  BULK COLLECT INTO name_list
  FROM person;
  FORALL i IN 1 .. name_list.count
    UPDATE person p
    SET p.salary = p.salary + 1000
    WHERE p.fname = name_list(i);
END;
/
SELECT * FROM person;
```

This shows the SELECT query results before the code:

FNAME	LNAME	SALARY
John	Smith	20000
Susan	Jones	30000
Mark	Blake	25000

Debra	Carlson	40000

And the SELECT query results after the code:

FNAME	LNAME	SALARY
John	Smith	21000
Susan	Jones	31000
Mark	Blake	26000
Debra	Carlson	41000

So that's how you can use both BULK COLLECT and FORALL to improve the performance of your PL/SQL code. If you want to learn more about these features, read this article: Bulk Processing with BULK COLLECT and FORALL.

Conclusion

Field and record types are handy features of PL/SQL and allow you to declare variables based on the type of a column or an entire row in a database table. They improve the maintainability of your code.

PL/SQL also allows you to use collections. These are similar to arrays but have some other advantages. The two types of collections are Index By Table and Nested Table.

Running SELECT statements or other DML statements individually can impact the performance of your code. It's better to switch between PL/SQL and SQL as little as possible. The BULK COLLECT and FORALL features allow you to do that in a better way.

Quiz

Question 1

What does %TYPE let you do?

- Nothing, it's not a valid command in PL/SQL.
- It lets you declare a variable that is based on the data type of a column.
- It lets you declare a variable that is based on the data type of all columns in a table.
- It lets you define a variable as being entered or typed in by the user.

Question 2

What's one difference between Index By Table and Nested Table?

- An Index By Table lets you specify the data type of the index.
- Nothing, they are both the same
- A Nested Table only works with nested SQL statements
- An Index By Table needs a table with an index on it.

Question 3

What's wrong with this code (assuming the emp_list exists and the employee table exists)?

```
FORALL i IN 1 .. emp_list.count LOOP
  UPDATE employee
  SET years = years + 1
  WHERE emp_name = emp_list.emp_name;
END LOOP;
```

- You can't run a FORALL statement on an UPDATE statement.
- The FORALL keyword doesn't exist.
- FORALL is not a loop so it should not have the LOOP and END LOOP statements.
- Nothing, it will run successfully.

Question 4

Why should you use BULK COLLECT?

- To reduce the switching between PL/SQL code and SQL code.
- You shouldn't use it at all as it's slow.
- To get data from large tables, but not small tables.
- It's required to declare collection variables.

Chapter 7: Nested Blocks and Packages

In this section, you'll learn:

- What nested blocks are and how to create them
- What packages are, why you should use them, and how to create them

What is a Nested Block in PL/SQL?

A nested block is where a block of PL/SQL code is nested inside another block. A PL/SQL block is a set of code inside the DECLARE, BEGIN, and END keywords. Inside the BEGIN and END keywords, you can place another block, which is the nested block.

Why would you do this? There are a few reasons:

- To keep the code focused on one area. A nested block may contain the code and logic for a smaller part of your overall code.
- To handle exceptions separately. If there is an issue in part of your code, you may want to handle that separately and then continue with the execution of your main code.

A standard PL/SQL block looks like this:

```
DECLARE
  your_variables;
BEGIN
  your_code;
END;
```

A nested block goes inside the BEGIN and END keywords:

```
DECLARE
  your_variables;
BEGIN
  your_code;
  [DECLARE
    more_variables;]
  BEGIN
    more_code;
  END;
  even_more_code;
END;
```

Your second BEGIN and END block is inside the first BEGIN and END block. You can have code before and after this nested block, and code inside the nested block. The DECLARE section inside the nested block is optional.

Example: Nested Block

Let's see an example of a nested block.

```
DECLARE
  l_salary person.salary%TYPE;
  l_fname person.fname%TYPE := 'John';
BEGIN
  SELECT salary
  INTO l_salary
  FROM person
  WHERE fname = l_fname;

  BEGIN
    UPDATE person
    SET salary = l_salary + 5000
    WHERE fname = l_fname;
    DBMS_OUTPUT.PUT_LINE('Salary updated.');
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error updating table: ' || SQLERRM);
  END;
  DBMS_OUTPUT.PUT_LINE('More code here.');
END;
```

In this code, we are selecting a salary value for John into a variable called l_salary.

Then, inside the nested block, we are running an UPDATE statement to update the salary, and then writing a message. If this nested block fails, then we write a message explaining the error that happened.

This is done so that the UPDATE statement can run and any errors are handled separately. If there is an issue updating the data, an error is shown, but the rest of the code keeps running.

Here's what the output shows:

```
1 row(s) updated.

Salary updated.

More code here.
```

If we change the code to make an error occur on the UPDATE statement, this is what happens:

```
DECLARE
  l_salary person.salary%TYPE;
  l_fname person.fname%TYPE := 'John';
BEGIN
  SELECT salary
  INTO l_salary
  FROM person
  WHERE fname = l_fname;

  BEGIN
    UPDATE person
    SET salary = l_salary + 5000
    WHERE fname = l_fname;
    DBMS_OUTPUT.PUT_LINE('Salary updated.');
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error updating table: ' || SQLERRM);
  END;
  DBMS_OUTPUT.PUT_LINE('More code here.');
END;
```

The output is shown:

```
1 row(s) updated.

Error updating table: ORA-01438: value larger than specified precision
allowed for this column
```

```
More code here.
```

This error is shown because the UPDATE statement encountered an error. The error was handled by an exception and written to the screen. The rest of the code kept running (the put_line function that wrote "More code here").

So that's how you can write and use nested blocks.

Let's take a look at another concept in PL/SQL: packages.

What is a Package in PL/SQL?

A package in PL/SQL is an object that groups logically related PL/SQL code. They have two parts: a package specification that defines the public interface for the package, and a package body which contains the code required for each procedure inside the package.

If you have experience in other programming languages, you can think of a package as being similar (but not the same as) a library. It contains multiple related procedures and the code can be called from other PL/SQL code.

Earlier in this guide, we just learned how to create procedures and functions. These objects are created on the database, but are standalone objects and are not inside packages. We'll learn how to create packages in this guide.

Why use Packages?

There are several reasons why you should use packages for your PL/SQL code:

Easier to design applications: You can write the specification first (the names and parameters) of the packages and procedures, and work on the body (the implementation details or code) later, if you like.

Better performance: Package code is loaded into memory when the code is first run. Whenever any other code in the package is run, it's accessed from memory. This means it's faster than reading and running the code from the disk.

Hide the details: You can provide access to the specification (procedure names and parameters) without providing the details of how they work. You can change the details whenever you want, and the code that refers to the package does not need to change.

Modular development: You can combine all of the related code into a package to make it easier to develop your application. Interfaces between packages can be developed and each package can be self-contained, similar to how classes work in object-oriented programming in other languages.

A package in PL/SQL contains two parts: a package specification and a package body. Let's take a look at what these are.

What is a Package Specification?

A package specification in PL/SQL is where public code is declared for a package. It contains any variables, functions, and procedures that you want others to know about and access. The functions and procedures contain the parameter names and types. This is so other users and programs can access them and run them, without knowing the details of how they work.

How can you create a package specification?

You use the CREATE PACKAGE statement. It contains the name of the package and any code you want to include in the package specification.

For example, a package for calculating customer order shipping information could be:

```
CREATE PACKAGE order_shipping AS

  PROCEDURE calc_shipping(dest_country VARCHAR2);

END order_shipping;
```

This code shows a few things:

- A new package is created, called order_shipping.
- It contains a single procedure called calc_shipping.
- The procedure takes one parameter: dest_country.

If you're calling this procedure, this is all you need to know. You pass the value for the country and the shipping is calculated.

What is a Package Body?

A package body contains the implementation details, or code, that the package specification requires. This includes the code for each of the procedures or functions, and details of cursors. This is where the code inside each of the procedures and functions go.

A package body is created with the CREATE PACKAGE BODY statement. The declarations of the objects in the body must match those of the specification (procedure names, parameters).

Following on from the earlier example, we can write a package body that details what the calc_shipping function does:

```
CREATE PACKAGE BODY order_shipping AS

  PROCEDURE calc_shipping(dest_country VARCHAR2) AS

  l_fee shipping_lookup.shipping_fee%TYPE;

  BEGIN

    SELECT shipping_fee

    INTO l_fee

    FROM shipping_lookup
    WHERE source_country = 'USA'

    AND destination_country = dest_country;


    DBMS_OUTPUT.PUT_LINE('Shipping fee is ' || l_fee);


  END;


END order_shipping;
```

This code looks up the shipping fee from a table and displays it on the screen. If you run this code, it will create the package body for this package.

Note: If you want to drop a package to make changes and create it again, run DROP PACKAGE package_name.

Example of Using a PL/SQL Package

We have a package and package body declared and created on the database, after running the code above.

Let's set up the sample data, and run the package code.

```
CREATE TABLE shipping_lookup (

  source_country VARCHAR2(100),

  destination_country VARCHAR2(100),

  shipping_fee NUMBER(10,2)

);

INSERT ALL

INTO shipping_lookup (source_country, destination_country, shipping_fee)
VALUES ('USA', 'USA', 5)

INTO shipping_lookup (source_country, destination_country, shipping_fee)
VALUES ('USA', 'UK', 18)

INTO shipping_lookup (source_country, destination_country, shipping_fee)
VALUES ('USA', 'France', 20)

INTO shipping_lookup (source_country, destination_country, shipping_fee)
VALUES ('USA', 'Canada', 7)

SELECT * FROM dual;
```

Now we can check the shipping_lookup table.

```
SELECT * FROM shipping_lookup;
```

SOURCE_COUNTRY	DESTINATION_COUNTRY	SHIPPING_FEE
USA	USA	5
USA	UK	18
USA	France	20
USA	Canada	7

We should have all we need to run our package code.

To run your package code, you refer to the package name and procedure name in PL/SQL:

```
BEGIN

  order_shipping.calc_shipping('UK');

END;
```

The output we get is

```
Statement processed.

Shipping fee is 18
```

This shows the shipping fee is 18, which is what our table shows. When we call the package, we don't care how the shipping fee is calculated, we only want to get the value.

Some improvements to this package could be:

- Return the shipping_fee value from the procedure, rather than show it using PUT_LINE.
- Add an exception section to handle values that are not found in the table.
- If the majority of shipping is done from USA but some is done from another country, perhaps add the source country as another parameter.

So that's how you can create a package in PL/SQL. First create the package object, then the package body with the details. The packages can then be run from any PL/SQL code block.

Conclusion

A nested block is a BEGIN/END block within another BEGIN/END block. They are useful for containing related code and for handling errors without impacting the entire program.

Packages are used to contain related functionality, such as variables and procedures. They help with application design and performance. A package specification contains the public interface, or what other users can see, such as the procedure names and parameters. The package body contains the details of the procedures, including the code inside them.

Quiz

Question 1

Can you have a BEGIN and END block within another BEGIN and END block?

- Yes, this is called a nested block.
- Yes, but only if both blocks contain a DECLARE section.
- Yes, but you can only have two at the most.
- No, this is not allowed in PL/SQL.

Question 2

What is the difference between a package specification and a package body?

- Nothing, they are two words that mean the same thing.
- A package specification is part of PL/SQL and a package body is not.
- A package specification contains the public information, such as function names. A package body contains the inner details of those functions.
- A package specification contains the details of functions, and a package body contains only the names and parameters.

Question 3

What's wrong with this code?

```
CREATE PACKAGE emp_calc AS
  PROCEDURE calc_bonus(emp_id NUMBER);
END emp_calc;
/
CREATE PACKAGE BODY emp_calc AS
  PROCEDURE calculate_bonus(emp_id NUMBER) AS
    BEGIN
      UPDATE emp
      SET bonus = salary * 0.1
      WHERE id = emp_id;
    END;
END emp_calc;
/
```

- Nothing, it works fine.
- You can't call an UPDATE statement inside a package.
- The procedure name in the body is different to the specification.
- You can't use parameters for procedures in packages.

Question 4

What happens if an error is found in a nested block, and it is handled by the EXCEPTION section and is passed to a DBMS_OUTPUT.PUT_LINE function?

- The error is raised in the main block as well, and if it is not handled, it will cause an error in the entire program.
- Nothing, the error is already handled so no extra treatment is needed.
- This is not valid as errors can't be handled in nested blocks.
- The error is raised in the main block and causes an error there, regardless of what code exists.

Conclusion

Thanks for reading and following along with this PL/SQL tutorial. I hope it's been helpful to you in learning PL/SQL. It covers all of the basics, and is enough to get you started with writing PL/SQL code and working with applications that use this code.

Thanks again!

Ben Brumm

www.DatabaseStar.com

Quiz Answers

Here are the answers to the quiz questions in this guide. The bolded answers are the correct answers.

Chapter 1

Question 1:

What does PL/SQL stand for?

- Powerful Loads of Structured Query Language
- **Procedural Language Structured Query Language**
- Packaged Language Structured Query Language
- Printable Language Structured Query Language

Question 2:

What keyword is used to start the executable section of a PL/SQL program?

- START
- RUN
- **BEGIN**
- END

Question 3:

What is the built-in function used for displaying output to the screen?

- PRINTLN

- WRITE
- SAVE
- **PUT_LINE**

Question 4:

What's missing from this simple Hello World program?

```
BEGIN

DBMS_OUTPUT.PUT_LINE

END;
```

- There is no START keyword.
- **There is no text value or brackets for the PUT_LINE function.**
- Nothing, it will run successfully.
- There is no function called PUT_LINE.

Chapter 2

Question 1:

What keyword is used to start the section where variables are created?

- START
- BEGIN
- **DECLARE**
- VARIABLE

Question 2:

What do you need to include when declaring a variable?

- **The name of the variable and the data type are required.**
- The name, data type, and the value are all required
- The name of the variable is required.
- The name, data type, the value, and the word CONSTANT are all required.

Question 3:

What is a constant?

- An area in memory that stores a value for your program and can be changed in your program.
- **A variable in your program that cannot be changed once it is set.**
- A function that displays data to the screen.
- A way to join two values together into one.

Question 4:

What's wrong with this code?

```
DECLARE
 l_radius NUMBER(8);
BEGIN
 DBMS_OUTPUT.PUT_LINE('The diameter is ' || 2 * l_radius);
END;
```

- Nothing. It will run and display the value of 16, because l_radius is set to 8 and the function multiplies it by 2.
- **The code will run, but it won't display a number because the value of l_radius is not set.**
- The code will run and display a value of 2.
- The code will not run because you can't concatenate a text value with a number value.

Chapter 3

Question 1:

What is the keyword used as part of an IF statement that runs code if none of the provided conditions are met?

- **ELSE**
- OTHERWISE
- FINAL
- END

Question 2:

What does the FOR statement do?

- Initialises a variable
- Checks if a condition is true or false, and runs the code inside one time if it is true
- **Starts a loop and runs the code inside many times until the criteria is false**
- Sets up the data required to send an email

Question 3:

Which of the following pieces of code will correctly check that two conditions are true in an IF statement?

- IF (l_input > 10) (l_input < 20) THEN
- IF (l_input > 10 & l_input < 20) THEN
- IF (l_input > 10 < 20) THEN
- **IF (l_input > 10 AND l_input < 20) THEN**

Question 4:

What is wrong with this code?

```
DECLARE
  l_loop_counter NUMBER(3) := 8;
BEGIN
   WHILE (l_loop_counter < 5) LOOP
    DBMS_OUTPUT.PUT_LINE('Loop number: ' || l_loop_counter);
    l_loop_counter := l_loop_counter + 1;
  END LOOP;
END;
```

- **It will run but no output is shown.**
- Nothing, it will run successfully and show five lines of output.
- It will run but cause an infinite loop.
- It will run and show eight lines of output.

Chapter 4

Question 1

Which one of these statements is true about the differences between a procedure and a function?

- They are both the same.
- **Functions must have a RETURN value, but procedures don't need one.**
- Procedures can only take IN parameters and not OUT parameters.
- Procedures can be used in regular SQL, but functions cannot.

Question 2

What does the OR REPLACE keyword do as part of the CREATE PROCEDURE or CREATE FUNCTION statement?

- **It replaces the existing object with the one you have defined in the statement if it already exists.**
- Nothing, this keyword is not valid.
- Nothing, it's a valid keyword but does not apply to procedures or functions.
- It determines if your code will insert new data into a table or update existing data.

Question 3

What is an exception?

- An error that happens when you create your procedure or function, such as a missing bracket.
- The part of an IF statement that runs if no criteria are met.
- **An error that happens while your code is running.**
- A collection of data that is not inserted into a table.

Question 4

How can you run the code in a procedure?

- **Run the "CALL procedure_name" statement.**
- It's run automatically when you create the procedure.
- Run the "RUN procedure_name" statement.

- You can't run procedures, you can only run functions.

Chapter 5

Question 1

How can you declare an explicit cursor?

- **cursor_name IS select_query;**
- cursor_name := select_query;
- cursor_name IS TYPE OF explicit_cursor;
- You can't, because explicit cursors are created automatically by Oracle.

Question 2

What does the %FOUND attribute of a cursor do?

- Returns the data from the cursor.
- Returns the number of rows found by the cursor
- Nothing, it won't work because it needs to be written in lowercase.
- **Returns true if at least one row is impacted by the query, otherwise it shows false**

Question 3

What does this code do?

```
TYPE id_arraytype IS VARRAY(10) OF NUMBER(4);

id_array id_arraytype;
```

- Nothing, it will show an error
- **It will declare a new type of an array of numbers, and a new variable of that type**
- It declares a new type of array, but does not declare a variable of that type
- It creates a new table with two columns, one as an array and the other as a number.

Question 4

How can you get the number of rows impacted by an INSERT statement in your PL/SQL code?

- **Use the sql%ROWCOUNT attribute.**
- SELECT from the table to find rows inserted today.

- You can't.
- Use the sql%NUMROWS attribute.

Chapter 6

Question 1

What does %TYPE let you do?

- Nothing, it's not a valid command in PL/SQL.
- **It lets you declare a variable that is based on the data type of a column.**
- It lets you declare a variable that is based on the data type of all columns in a table.
- It lets you define a variable as being entered or typed in by the user.

Question 2

What's one difference between Index By Table and Nested Table?

- **An Index By Table lets you specify the data type of the index.**
- Nothing, they are both the same
- A Nested Table only works with nested SQL statements
- An Index By Table needs a table with an index on it.

Question 3

What's wrong with this code (assuming the emp_list exists and the employee table exists)?

```
FORALL i IN 1 .. emp_list.count LOOP
  UPDATE employee
  SET years = years + 1
  WHERE emp_name = emp_list.emp_name;
END LOOP;
```

- You can't run a FORALL statement on an UPDATE statement.
- The FORALL keyword doesn't exist.
- **FORALL is not a loop so it should not have the LOOP and END LOOP statements.**

- Nothing, it will run successfully.

Question 4

Why should you use BULK COLLECT?

- **To reduce the switching between PL/SQL code and SQL code.**
- You shouldn't use it at all as it's slow.
- To get data from large tables, but not small tables.
- It's required to declare collection variables.

Chapter 7

Question 1

Can you have a BEGIN and END block within another BEGIN and END block?

- **Yes, this is called a nested block.**
- Yes, but only if both blocks contain a DECLARE section.
- Yes, but you can only have two at the most.
- No, this is not allowed in PL/SQL.

Question 2

What is the difference between a package specification and a package body?

- Nothing, they are two words that mean the same thing.
- A package specification is part of PL/SQL and a package body is not.
- **A package specification contains the public information, such as function names. A package body contains the inner details of those functions.**
- A package specification contains the details of functions, and a package body contains only the names and parameters.

Question 3

What's wrong with this code?

```
CREATE PACKAGE emp_calc AS

  PROCEDURE calc_bonus(emp_id NUMBER);
```

```
END emp_calc;
/
CREATE PACKAGE BODY emp_calc AS
  PROCEDURE calculate_bonus(emp_id NUMBER) AS
    BEGIN
      UPDATE emp
      SET bonus = salary * 0.1
      WHERE id = emp_id;
    END;
END emp_calc;
/
```

- Nothing, it works fine.
- You can't call an UPDATE statement inside a package.
- **The procedure name in the body is different to the specification.**
- You can't use parameters for procedures in packages.

Question 4

What happens if an error is found in a nested block, and it is handled by the EXCEPTION section and is passed to a DBMS_OUTPUT.PUT_LINE function?

- The error is raised in the main block as well, and if it is not handled, it will cause an error in the entire program.
- **Nothing, the error is already handled so no extra treatment is needed.**
- This is not valid as errors can't be handled in nested blocks.
- The error is raised in the main block and causes an error there, regardless of what code exists.