1 Stored Procedures

Many modern databases support a more procedural approach to databases—they allow you to write procedural code to work with data. Usually, it takes the form of SQL interweaved with the more familiar `IF` statements and loops, etc.

Note that this has nothing to do with *accessing* the database. You can access any database from virtually any language. What we're talking about is the code that is executed by the database server.

While there are many various 'database' languages, we will only talk about the primary two: T-SQL, which is supported by SQL Server and Sybase, and PL/SQL, which is supported by Oracle.

Many other languages may be supported. For example, Oracle allows you to write stored procedures and triggers in Java, etc.

2 Oracle Users

Before you start working within this section, it's a good idea to create a user to play with. You can do that (in Oracle) via the 'create user' statement, ie:

```
SQL> create user blah identified by glah;

User created.
```

The above will create a user named 'blah' with password 'glah'. Once the user is created, they cannot login. They need to be granted the 'create session' priviledge, ie:

```
SQL> grant create session to blah;

Grant succeeded.
```

Now the user can login. All the tables they create will be in their user space (their own little database). You can view all of user's tables via:

```
SQL> select table_name from user_tables order by table_name;
```

To create stored procedures, the user should also have 'create procedure' priviledge, ie:

```
SQL> grant create procedure to blah;

Grant succeeded.
```

You can also alter a user's password via:

```
SQL> alter user blah identified by abc123;

User altered.
```

Also, if you're already logged in, you can change *your* password just by typing "password" at the sql prompt.

3 MSSQL Users

In Microsoft SQL Server^{TM}, if you want to create a restricted space for users to own, you have to give them their own database (unlike in Oracle, where you have one database).

```
DROP DATABASE blahdb
GO

CREATE DATABASE blahdb COLLATE SQL_Latin1_General_CP1_CI_AS
GO

USE blahdb
GO
```

Before we are scripting the database creation, it's a good idea to first drop any existing databases by that name (so that the creation succeeds). The "collation" parameter indicates what character set will be used for all character types in this database, and also how those characters will be compared. For example, we can set it up so store both lower case and upper case ascii characters, but to ignore case whenever doing any comparisons.

We can now add a login:

```
-- loginname, password, defaultdb
EXEC sp_addlogin 'blahuser', 'blah123', 'blahdb'
GO
```

the below adds a user 'blahuser', with password 'blah123', and sets their default login database to 'blahdb' (the database we created above).

To let the user 'own' a database (and have all privileges on it), we simply switch to the desired database, and change its owner:

```
USE blahdb
GO

EXEC sp_changedbowner 'blahuser'
GO
```

If you ever need to drop a user, you can do that with:

```
EXEC sp_droplogin 'blahuser'
GO
```

There are a few quicky bits: you cannot drop a user that owns a database. You have to first re-assign ownership (via the `sp_changedbowner`) of the database to someone else, then you can drop the user.

All of the above should be in `createdb.sql` file (or similarly clearly named) in every database application you produce. Meaning, you should be able to go from a fresh install of a database, to having your application setup and ready to run on that database by simply running one or two script files (messing with menu options—for possibly a slightly different release of a database; with all menu options re-shuffled for no good reason) looks really bad and unprofessional when you're doing it in front of a client.

4 Oracle PL/SQL

Besides plain vanilla SQL, Oracle supports PL/SQL. The PL stands for 'Procedural Language', which means you can have things like `IF` statements, loops, variables, and other procedural things along with *declarative* SQL statements.

4.1 Variables

Just as most procedural languages, PL/SQL has some sort of variables. The types of variables are plain SQL column types that you're all used to. You can also refer to a type of a particular column explicitly by specifying the fully qualified column name (`tablename.columname`) followed by `%TYPE`. For example: `PRODUCT.PRICE%TYPE`.

Similarly, we can refer to a particular row as a single type. Again, you declare it by referring to a database table directly: `PRODUCT%ROWTYPE`. This would refer to a single record stored in the PRODUCT table.

Along with the above mentioned, some common types are: `BOOLEAN`, `DATE`, `NUMBER`, `CHAR`, and `VARCHAR2`.

We declare variables of these types just like we specify columns in tables. First, we list the name of the variable, then the type we want it to have. For example, to declare a price variable of a fixed point `NUMBER`, we might do something like this:

```
PRICE NUMBER(6,2);
```

4.2 PL/SQL Program Blocks

PL/SQL programs are structured in blocks and have the following format:

```
DECLARE
    variable_declarations
BEGIN
    procedural_code
EXCEPTION
    error_handling
END;
```

4.2.1 Declare

The declare part is where variable declaration goes. All used variables must be declared in this section. This is also the place where other more exotic variable types are declared, like cursors and exceptions.

4.2.2 Begin

This is the part we're most interested in. This is where the bulk of your programs will be placed. Here, you can have `IF` statements, loops, etc.

4.2.3 Exceptions

The exception section is optinal, and is where we place error handling code. We will talk about it later.

4.2.4 End

The end signifies the end of this program block.

4.3 Operators

PL/SQL supports several operators to do various things. Table 1 lists some of the more common operators.

$**$	Exponentiation
$*$	Multiplication
$/$	Division
$+$	Addition
$-$	Subtraction
$-$	Negation
$:=$	Assignment
$=$	Equals Comparison
$<>$	Not Equals Comparison
$!=$	Not Equals Comparison
$>$	Greater Than Comparison
$<$	Less Than Comparison
$>=$	Greater Than or Equal Comparison
$<=$	Less Than or Equal Comparison
AND	The obvious AND operation
OR	The obvious OR operation
$:=$	Assignment
$\|\|$	String Concatenation

Table 1: PL/SQL Operators

4.4 Hello World and a bit Beyond...

Well, let's start with the PL/SQL 'hello world' example. Before we write it however, there are a couple of things that need to be setup. First, start "SQL*Plus" (Oracle), login, and type:

```
SET SERVEROUTPUT ON
```

What this does is enable you to view output in SQL*Plus window whenever your programs attempts to write some output to the screen. If you don't set it, the output won't appear in the SQL*Plus terminal.

Now, let's get on with the show. Type the following into the SQL*Plus window as is:

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

You'll notice that it doesn't run as an average SQL statement. To make it run, you must type the '/' character on a line by itself. And you'll notice:

```
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('Hello World');
  3  END;
  4  /
Hello World

PL/SQL procedure successfully completed.
```

You can think of `DBMS_OUTPUT.PUT_LINE()` as sort of a `printf()` in C language. It writes output to the console; but it only works for strings (or data that can be *implicitly* converted to a string).

You can also do more complicated things, like access global variables like `SYSDATE`. For example:

```
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('The time now is: ');
  3      DBMS_OUTPUT.PUT_LINE(SYSDATE);
  4  END;
  5  /
The time now is:
31-JUL-02
```

We're not done with this simple example yet. We can also modify the DATE format:

```
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('The time now is: ');
  3      DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE,'MM/DD/YYYY'));
  4  END;
  5  /
The time now is:
07/31/2002
```

4.5 Type Conversion Functions

From the previous example, you can see we've used the `TO_CHAR` function to format the date. Table 2 lists some of these useful functions.

4.6 Character String Functions

There are a number of functions for handling character string data in PL/SQL, these include the easy to use string catenation operator. For example, we could have written our time now is example as:

```
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE('The time now is: ' || SYSDATE);
  3  END;
  4  /
The time now is: 31-JUL-02
```

Note that `||` was used to concatenate the string `'The time is now:  '` with the `SYSDATE`. Some of the more useful PL/SQL are listed in Table 3.

Substring example follows:

```
SQL> SELECT SUBSTR('HELLO',2,4) FROM DUAL;

SUBS
----
ELLO
```

4.7 PL/SQL IF Statement

PL/SQL, being a procedural language naturally has lots of flow control constructs, from `IF` statements to `WHILE` loops.

Remember to type: `SET SERVEROUTPUT ON` in SQL*Plus before running any programs, so that you can see the output.

`TO_DATE`	Converts a string to a date.
`TO_NUMBER`	Converts a character string to a number.
`TO_CHAR`	Converts numbers or dates to character strings.

Table 2: Some PL/SQL Functions

`RTRIM(STR)`	Removes blank spaces from right side of string.
`LENGTH(STR)`	Returns the length of the string.
`UPPER(STR)`	Converts the string to upper case.
`LOWER(STR)`	Converts the string to lower case.
`INSTR(STR1,STR2)`	Looks for STR2 in STR1.
`SUBSTR(STR,START,END)`	Returns a substring that starts at START.

Table 3: More PL/SQL Functions

4.8 IF - THEN Structure

The general format of an IF statement is:

```
IF condition THEN
    program_statements
END IF;
```

Assuming we all know how to program, and know what `IF` statements are, I'm not going to spend too much time on the obvious.

An example program that uses an `IF` statement is:

```
DECLARE
    A NUMBER(6);
    B NUMBER(6);
BEGIN
    A := 23;
    B := A * 5;
    IF A < B THEN
        DBMS_OUTPUT.PUT_LINE('Ans: ' || A || ' is less than ' || B);
    END IF;
END;
```

Which produces the expected output of:

```
Ans: 23 is less than 115
```

4.9 IF - ELSE Structure

Just as in any programming language that has an `IF` statement, there is also the `ELSE` clause to the `IF` statement. The full structure of an `IF` statement is thus:

```
IF condition THEN
    if_condition_is_true_code
ELSE
    if_condition_is_false_code
END IF;
```

Let's modify our simple example to:

```
DECLARE
    A NUMBER(6);
    B NUMBER(6);
BEGIN
    A := 23;
    B := A / 5;
    IF A < B THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Ans: ' || A || ' is less than ' || B);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Ans: ' || A || ' is greater than ' || B);
    END IF;
END;
```

Note that we've also modified the `B := A * 5` to `B := A / 5` in order to test the `ELSE` condition.

4.10 IF Statement nesting

We can also put `IF` statements inside other `IF` statements. Here, again, let's jump right into an example:

```
DECLARE
    A NUMBER(6);
    B NUMBER(6);
    C NUMBER(6);
    ABCMAX NUMBER(6);
BEGIN
    A := 23;
    B := A / 5;
    C := B * 7;
    IF A > B THEN
        IF A > C THEN
            ABCMAX := A;
        ELSE
            ABCMAX := C;
        END IF;
    ELSE
        IF B > C THEN
            ABCMAX := B;
        ELSE
            ABCMAX := C;
        END IF;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Max of: ' || A || ', ' || B ||
        ', and ' || C || ' is ' || ABCMAX);
END;
```

The code above finds the maximum value (`ABCMAX`) of three variables (`A`, `B`, and `C`).
The code looks self explanatory; so we won't spend much time on it.

4.11 IF ELSIF Structure

When `IF` and `ELSE` are not enough, we can resort to using `ELSIF`. This is an else if equivalent in C (and in Perl it is actually named `elsif`).

Let’s say we wanted to calculate the letter grade given a number grade, we may write a program such as:

```
DECLARE
    NGRADE NUMBER;
    LGRADE CHAR(2);
BEGIN

    NGRADE := 82.5;

    IF NGRADE > 95 THEN
        LGRADE := 'A+';
    ELSIF NGRADE > 90 THEN
        LGRADE := 'A';
    ELSIF NGRADE > 85 THEN
        LGRADE := 'B+';
    ELSIF NGRADE > 80 THEN
        LGRADE := 'B';
    ELSIF NGRADE > 75 THEN
        LGRADE := 'C+';
    ELSIF NGRADE > 70 THEN
        LGRADE := 'C';
    ELSIF NGRADE > 65 THEN
        LGRADE := 'D+';
    ELSIF NGRADE > 60 THEN
        LGRADE := 'D';
    ELSE
        LGRADE := 'F';
    END IF;
    DBMS_OUTPUT.PUT_LINE('Grade ' || NGRADE || ' is ' || LGRADE);
END;
```

Which for our particular example number grade produces output:

```
Grade 82.5 is B
```

4.12 PL/SQL - SQL

Apparently, we can run SQL statements inside PL/SQL! Isn’t that amazing?

We can’t use all of SQL though, we can only use DML (Data Manipulation Language) which includes statements like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, and transaction control statements, like `COMMIT`, `ROLLBACK`, `SAVEPOINT`.

The only limitation seems to be are DDL statements, which are used to `CREATE`, `ALTER`, and `DROP` tables, and `GRANT` privileges, just to name a few.

4.12.1 Simple Example

For right now, here's a simple example. We'll do more as we learn PL/SQL. In this example, we'll insert a new PRODUCT into a simple database.

```
DECLARE
    PID NUMBER(6);
BEGIN
    PID := 20;
    INSERT INTO product VALUES (PID,'tv',32,199.99);
    PID := PID + 1;
    INSERT INTO product VALUES (PID,'vcr',16,799.98);
    COMMIT;
END;
```

We can now run a `SELECT` statement to retrieve the values we've inserted:

```
SELECT * FROM PRODUCT WHERE PRODUCT_ID >= 20;
```

Which produces the expected results:

```
PRODUCT_ID DESCRIPTION
---------- ------------
        20 tv
        21 vcr
```

Notice that in our example, we used a variable named `PID` inside our `INSERT` statement. That's the real power of PL/SQL, where we can use procedural language constructs and variables to drive our database SQL code. PL/SQL Loops

Just as with `IF` statements, PL/SQL also has loops. Loops are used to repeat some action multiple times, until some condition is met.

PL/SQL has five looping structures, and we shall talk about each one in more depth as we move along. So without further interruption, I present to you...

4.13 LOOP ... EXIT Loop

The general format of such a loop is:

```
LOOP
    various_statements
    IF condition THEN
        EXIT;
    END IF;
    various_statements
END LOOP;
```

This loop is very similar to an infinite loop in C/C++, where you use break; to terminate the loop; in this case, the `EXIT;` command takes the form of break.

Note that we can place various program statements before the exiting `IF` statement and after, which gives us great flexibility about when and how the condition is evaluated.

An example of such a loop would be:

```
DECLARE
    I NUMBER(6);
BEGIN
    I := 1;
    LOOP
        DBMS_OUTPUT.PUT_LINE('aI: ' || I);

        I := I + 1;
        IF I > 5 THEN
            EXIT;
        END IF;

        DBMS_OUTPUT.PUT_LINE('bI: ' || I);
    END LOOP;
END;
```

With the expected output of:

```
aI: 1
bI: 2
aI: 2
bI: 3
aI: 3
bI: 4
aI: 4
bI: 5
aI: 5
```

Note that you should `SET SERVEROUTPUT ON;` in order to see the output in SQL*Plus screen.

Also, it would be *very* helpful if you trace the above program to ensure that you understand how the loop functions and why the results look as they do. I shall not provide the output for the following code, and expect you to run it yourself.

4.14 LOOP ... EXIT WHEN Loop

To simplify our writing our the `IF` statement, there is a simpler form, the `EXIT WHEN` loop. The general format of such a loop is:

```
LOOP
    various_statements
    EXIT WHEN condition;
    various_statements
END LOOP;
```

An example usage of such a loop would be something like this:

```
DECLARE
    I NUMBER(6);
BEGIN
    I := 1;
    LOOP
        DBMS_OUTPUT.PUT_LINE('aI: ' || I);
        I := I + 1;
        EXIT WHEN I > 5;
        DBMS_OUTPUT.PUT_LINE('bI: ' || I);
    END LOOP;
END;
```

You should run this code yourself. It would actually be more helpful if you write out the output first, and then compare it to the actual results.

4.15 WHILE ... LOOP Loop

Our next loop is the all familiar `WHILE` loop, except now it is in PL/SQL and not in C/C++. It works nearly identically though. The idea is that you have a condition which is tested each time through the loop, and if it's false, the loop terminates.

The general format of such a loop is:

```
WHILE condition
LOOP
    various_statements
END LOOP;
```

Our typical (as in typical for these class notes) would be:

```
DECLARE
    I NUMBER(6);
BEGIN
    I := 1;
    WHILE I <= 5
    LOOP
        DBMS_OUTPUT.PUT_LINE('aI: ' || I);
        I := I + 1;
        DBMS_OUTPUT.PUT_LINE('bI: ' || I);
    END LOOP;
END;
```

Just as with the previous code, you should try to figure out what the output is, and then run it to see if your trace was correct. Tracing questions such as these are fair game for quizzes and tests.

4.16 FOR Loop

There is also the traditional numeric FOR loop that's commonly found in most procedural languages. The general format of such a loop is:

```
FOR countervariable IN startvalue .. endvalue
LOOP
    various_statements
END LOOP;
```

The start and end values must be integers, and are always incremented by one. An example of such a loop would be:

```
BEGIN
    FOR I IN 1..5
    LOOP
        DBMS_OUTPUT.PUT_LINE('I: ' || I);
    END LOOP;
END;
```

Notice that we never actually directly initialize `I`, or even declare it! It is done implicitly for us by the `FOR` loop. You should run this code to ensure you understand it.

You can also use other variables to loop on. For example, to loop from J to K, you'd do something like:

```
DECLARE
    J NUMBER(6);
    K NUMBER(6);
BEGIN
    J := 7;
    K := 2;
    FOR I IN K..J
    LOOP
        DBMS_OUTPUT.PUT_LINE('I: ' || I);
    END LOOP;
END;
```

Again, notice that we never actually initialize nor declare `I`. In fact, the `I` in the loop is a totally different variable. Even if you have an `I` variable declared, the loop will still use its own version. You can verify that by running this code:

```
DECLARE
    I NUMBER(6);
BEGIN
    I := 7;
    DBMS_OUTPUT.PUT_LINE('BEFORE LOOP I: ' || I);
    FOR I IN 1..5
    LOOP
        DBMS_OUTPUT.PUT_LINE('IN LOOP I: ' || I);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('AFTER LOOP I: ' || I);
END;
```

Which interestingly enough, prints out:

```
BEFORE LOOP I: 7
IN LOOP I: 1
IN LOOP I: 2
IN LOOP I: 3
IN LOOP I: 4
IN LOOP I: 5
AFTER LOOP I: 7
```

Which illustrates that the value of our declared variable `I` is unchanged by the loop (and that the loop internally has I declared which is different from our explicitly declared `I`).

4.17 Cursors

Before we move on with our discussion of the next and last loop construct, we must cover the concept of Cursors.

Oracle has two major different types of cursors. One is implicit and the other one is explicit.

4.18 Implicit Cursor

Implicit cursors can be generated every time you do a `SELECT` statement in PL/SQL. The general format goes something like this:

```
SELECT selectfields INTO declared_variables FROM table_list WHERE search_criteria;
```

The only catch is that the search criteria must return one and only one result. If it returns zero, or more than one, an error is generated.

For example, lets say we wanted to get the name and price of some specific product (identified by `PRODUCT_ID`):

```
DECLARE
    NAME PRODUCT.DESCRIPTION%TYPE;
    AMOUNT PRODUCT.PRICE%TYPE;
BEGIN
    SELECT DESCRIPTION,PRICE INTO NAME, AMOUNT
        FROM PRODUCT WHERE PRODUCT_ID = 4;
    DBMS_OUTPUT.PUT_LINE('PRICE OF ' || NAME || ' IS ' || AMOUNT);
END;
```

Which faithfully displays out:

```
PRICE OF keyboard IS 19.95
```

Assuming the "keyboard" is in the database and has `PRODUCT_ID = 4` (and has that price).

Note that we used the table's types, which brings up another issue: Now is a pretty good time to illustrate the ROWTYPE type. Let's rewrite the above using that.

```
DECLARE
    P PRODUCT%ROWTYPE;
BEGIN
    SELECT * INTO P FROM PRODUCT WHERE PRODUCT_ID = 4;
    DBMS_OUTPUT.PUT_LINE('PRICE OF ' || P.DESCRIPTION || ' IS ' || P.PRICE);
END;
```

Notice that the code got a lot smaller since we don't have to worry about defining every single variable for retrieval purposes. We retrieve a whole row of data at a time. The output of the above code is exactly the same as the previous.

4.19 Explicit Cursor

Explicit Cursors are cursors that you have to explicitly declare, and which give you a lot more flexibility than the implicit ones.

To declare an explicit cursor, you have to do it in the `DECLARE` section. The format looks something like:

```
CURSOR cursorname IS SELECT_statement;
```

Where `SELECT_statement` is any select statement (except a more exotic one which contains a `UNION` or `MINUS`.

5 Opening an Explicit Cursor

In order to use an explicit cursor, you must open it. You do that with a simple:

```
OPEN cursorname;
```

(obviously you have to do that inside the code section, between `BEGIN` and `END`).

5.1 Fetching Data into an Explicit Cursor

Besides opening the cursor, we also have to grab the results of the SELECT statement one by one. We do that with a FETCH. For example:

```
FETCH cursorname INTO recordvariables;
```

We shall do some examples when we learn our cursor loops, so hang on...

5.2 Closing a Cursor

Closing a cursor is just as easy as opening it. We just say:

```
CLOSE cursorname;
```

Cursors will be closed automatically once your code exits, but it's still a good idea to close them explicitly.

5.3 LOOP ... EXIT WHEN Loop (Again)

We can use our standard loops in order to go loop through the results returned by the cursor. So, let's move on to our example:

```
DECLARE
    P PRODUCT%ROWTYPE;
    CURSOR PRODUCTCURSOR IS
        SELECT * FROM PRODUCT;
BEGIN
    OPEN PRODUCTCURSOR;
    LOOP
        FETCH PRODUCTCURSOR INTO P;
        EXIT WHEN PRODUCTCURSOR%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('PRICE OF ' || P.DESCRIPTION || ' IS ' || P.PRICE);
    END LOOP;
    CLOSE PRODUCTCURSOR;
END;
```

Go through the code line by line. First, we declare our P variable which is a ROWTYPE from table PRODUCT. We then declare our CURSOR, which simply selects everything from the PRODUCT table.

Our code then proceeds to OPEN the cursor. We then fall into our standard loop (which we learned about earlier), and FETCH results from the CURSOR. We EXIT the loop if we got no more results (the PRODUCTCURSOR%NOTFOUND condition). If we did not exit the loop, we output product description and product price.

In the end, we just CLOSE the cursor. Depending on what you have in your PRODUCT table, the results of the code may look similar to this:

```
PRICE OF mice IS 26.99
PRICE OF keyboard IS 19.95
PRICE OF monitor IS 399.99
PRICE OF speakers IS 9.99
PRICE OF stapler IS 14.99
PRICE OF calculator IS 7.99
PRICE OF quickcam IS 99.98
PRICE OF harddrive IS 199.99
PRICE OF tv IS 199.99
PRICE OF vcr IS 799.98
```

You should go through the code, trace it, run it, and make sure you understand it.

5.4 Cursor Attributes

We've already seen one of the more important cursor attributes, the `%NOTFOUND`. There are also these:

`%NOTFOUND`: Evaluates to `TRUE` when cursor has no more rows to read. FALSE otherwise.

`%FOUND`: Evaluates to `TRUE` if last `FETCH` was successful and `FALSE` otherwise.

`%ROWCOUNT`: Returns the number of rows that the cursor has already fetched from the database.

`%ISOPEN`: Returns `TRUE` if this cursor is already open, and `FALSE` otherwise.

5.5 Cursor FOR ... IN ... LOOP Loop

There is also a special loop structure made specifically for working with cursors. It allows for easier cursor handling; it opens and closes the cursor for us, and we don't have to explicitly check for the end.

It is a `for` loop that has the general format:

```
FOR variable(s) IN cursorname LOOP
    various_program_statements
END LOOP;
```

Let us rewrite our example program (presented earlier) to use this new type of loop:

```
DECLARE
    P PRODUCT%ROWTYPE;
    CURSOR PRODUCTCURSOR IS
        SELECT * FROM PRODUCT;
BEGIN
    FOR P IN PRODUCTCURSOR LOOP
        DBMS_OUTPUT.PUT_LINE('PRICE OF ' || P.DESCRIPTION || ' IS ' || P.PRICE);
    END LOOP;
END;
```

Notice that the code got quite a bit simpler, with lots of cursor handling code gone; which is now being handled by the loop itself.

If you're really into optimization, you might want to improve the above code not to return the whole `%ROWTYPE` but individual fields which we're displaying, for example:

```
DECLARE
    CURSOR PRODUCTCURSOR IS
        SELECT DESCRIPTION,PRICE FROM PRODUCT;
BEGIN
    FOR P IN PRODUCTCURSOR LOOP
        DBMS_OUTPUT.PUT_LINE('PRICE OF ' || P.DESCRIPTION || ' IS ' || P.PRICE);
    END LOOP;
END;
```

Notice several things about the code: that we no longer declare `P` which is used for loop purposes. Also notice that our cursor is no longer returning everything, but just two individual fields which we're displaying.

5.6 Introduction to Stored Procedures

Just like any other procedural language, PL/SQL has code fragments that are called `PROCEDURES`.

You can call these `PROCEDURES` from other code fragments, or directly from SQL*Plus (or some other client program).

Before you begin to write procedures though, you need to verify that you have enough privileges to do that. If you don't (which probably means you're using a plain user account), then you need to login as administrator (or ask the administrator) to grant you access. To grant such priviledge yourself (in case you're the administrator - running Oracle on your own machine) you can do:

```
GRANT CREATE PROCEDURE TO someusername;
```

From that point on, the user someusername will be allowed to create, drop, and replace procedures and functions.

5.7 PROCEDURES

Procedures are code fragments that don't normally return a value, but may have some outside effects (like updating tables). The general format of a procedure is:

```
PROCEDURE procedure_name IS
BEGIN
    procedure_body
END;
```

Of course, you'll usually be either creating or replacing the procedure, so you'd want to add on `CREATE` (`OR REPLACE`) to the declaration. For example, to create (or replace) a `HELLO` procedure, you might do something like this:

```
CREATE OR REPLACE
PROCEDURE HELLO IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

The above declares a `HELLO` procedure that just displays `'Hello World'`. You can run it as part of a code fragment, or inside other procedures (or functions). For example:

```
BEGIN
    HELLO();
END;
```

Or you can simply execute it in SQL*Plus by typing:

```
CALL HELLO();
```

5.8 General Format

The general format of a create procedure statement is this:

```
CREATE OR REPLACE
PROCEDURE procedure_name ( parameters ) IS
BEGIN
    procedure_body
END;
```

Where `procedure_name` can be any valid SQL name, parameters is a list of parameters to this procedure (we'll discuss them later), and `procedure_body` is various PL/SQL statements that make up the logic of the procedure.

5.8.1 Parameters

The parameters (or arguments) are optional. You don't have to specify anything (not even the parenthesis). For example, a sample procedure, which you no doubt have already seen:

```
CREATE OR REPLACE
PROCEDURE HELLOWORLD IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
```

Never actually defines any parameters. What's the use of a procedure that doesn't take any parameters and doesn't return anything? Well, you may be interested in the procedure's side effects, like in our case, we're interested in our procedure displaying 'Hello World!' and nothing else. There may be many instances where you may want to just do something to the database, without any particular parameters, and without returning anything.

Anyway, this section is about parameters so let's talk about parameters.

Parameters are defined in a similar way as in a `CREATE TABLE` statement, which is similar to how variables are declared. You first specify the name of the variable, and then the type. For example:

```
 (N INT)
```

Would setup some procedure to accept an `INT` variable named `N`. Writing a simple procedure to display a variable name, you can come up with something like this:

```
CREATE OR REPLACE
PROCEDURE DISPN (N INT) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('N is ' || N);
END;
```

Which if you call, will promptly display:

```
SQL> CALL DISPN(1234567891);
N is 1234567891
```

You can also have multiple parameters. For example, you can accept A and B and display their sum and product.

```
CREATE OR REPLACE
PROCEDURE DISP_AB (A INT, B INT) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('A + B = ' || (A + B));
    DBMS_OUTPUT.PUT_LINE('A * B = ' || (A * B));
END;
```

Which when ran, displays something like (depending on the values you provide):

```
SQL> CALL DISP_AB(17,23);
A + B = 40
A * B = 391
```

Btw, it should be noted that you can use any PL/SQL type as an argument. For example, `VARCHAR` and others are perfectly acceptable. For example:

```
CREATE OR REPLACE
PROCEDURE DISP_NAME (NAME VARCHAR) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hi ' || NAME || '!');
END;
```

Which when called displays:

```
SQL> CALL DISP_NAME('John Doe');
Hi John Doe!
```

5.9 IN, OUT, IN OUT

There are various different parameter varieties (not types). For example, for the time being, we've only been giving the procedure data via parameters. This is the default (`IN`).

What we could also do is get data from the procedure, via an `OUT` parameter. To do that, we simply specify `OUT` in between the parameter name and its type. For example:

```
CREATE OR REPLACE
PROCEDURE SUM_AB (A INT, B INT, C OUT INT) IS
BEGIN
    C := A + B;
END;
```

Notice that the above code does not display the resulting sum, it just changes the value of the `C` parameter. Also notice the word `OUT` right after the declaration of `C` parameter name.

Anyway, we will use a code fragment to call the procedure:

```
DECLARE
    R INT;
BEGIN
    SUM_AB(23,29,R);
    DBMS_OUTPUT.PUT_LINE('SUM IS: ' || R);
END;
```

Which when ran, displays:

```
SUM IS: 52
```

Notice how we called the procedure with an argument to eventually retrieve the `OUT` result.

There is also the other special way of passing parameters: `IN OUT`. What that means is that we first can read the parameter, then we can change it. For example, we can write a procedure that doubles a number:

```
CREATE OR REPLACE
PROCEDURE DOUBLEN (N IN OUT INT) IS
BEGIN
    N := N * 2;
END;
```

To run it, we also create a small code fragment:

```
DECLARE
    R INT;
BEGIN
    R := 7;
    DBMS_OUTPUT.PUT_LINE('BEFORE CALL R IS: ' || R);
    DOUBLEN(R);
    DBMS_OUTPUT.PUT_LINE('AFTER CALL R IS: ' || R);
END;
```

Which when ran displays:

```
BEFORE CALL R IS: 7
AFTER CALL R IS: 14
```

Notice how this particular call first grabbed the value of a parameter, then set it in order to return the double of the value.

You can generally intermix these various ways of passing parameters (along with various types). You can use these to setup return values from procedures, etc.

5.10 Dropping Procedures

If you're interested in getting rid of a procedure totally, you can DROP it. The general format of a DROP is:

```
DROP PROCEDURE procedure_name;
```

That's all there is to stored procedures. We will do some practice exercises and more experimentation, but overall, that's all there is to them.

5.11 Functions

Functions are special types of procedures that have the capability to return a value.

It is a very shady question of when to use what, either functions or procedures. A good rule of thumb is: if you're interested in the "results" of the code, then you use a function, and return those results. If you're interested in the "side effects" (like table updates, etc.) and not about the "result" when you should use a procedure. Usually it doesn't affect your code all that much if you use a procedure or a function.

5.12 General Format

The general format of a function is very similar to the general format of a procedure:

```
CREATE OR REPLACE
FUNCTION function_name (function_params) RETURN return_type IS
BEGIN
    function_body
    RETURN something_of_return_type;
END;
```

For example, to write a function that computes the sum of two numbers, you might do something like this:

```
CREATE OR REPLACE
FUNCTION ADD_TWO (A INT,B INT) RETURN INT IS
BEGIN
    RETURN (A + B);
END;
```

To run it, we'll write a small piece of code that calls this:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('RESULT IS: ' || ADD_TWO(12,34));
END;
```

Which procudes the output:

```
RESULT IS: 46
```

All of a sudden, we know how to make functions (since we already know how to crate procedures). That's really all there is to it.

5.13 Dropping Functions

To drop a function, you do it in a similar way to a procedure. You simply say:

```
DROP FUNCTION function_name;
```

Oh, btw, to display the list of procedures/functions or plain general user objects that you have you can run a query:

```
SELECT OBJECT_NAME
FROM USER_OBJECTS
WHERE OBJECT_TYPE = 'FUNCTION';
```

You can do a similar thing for procedures.

6 Stored Procedures in T-SQL

Stored Procedures in any database work mostly the same—and in most cases, look and 'feel' the same. For example, when writing stored procedures in T-SQL, we're doing nearly identical things as in PL/SQL, with slight differences. What this means is that whatever you learn from one database is—with slight syntactical modifications—is transferable to another database.

7 T-SQL

Transact-SQL is the SQL flavor used by Microsoft SQL Server. As far as basic SQL goes, it is similar to Oracle, etc., as far as stored procedures are concerned, it is slightly different, but still similar. For example, a stored procedure to select all “Person” records may look like:

```
CREATE PROCEDURE GetAllPeople
AS
    SELECT *
    FROM Person
```

Just like in PL/SQL, you can group many statements with a `BEGIN` and `END` keywords—for situations where you want to do more than one statement.

7.1 Parameters

Parameters are passed to stored procedures by listing them right after the procedure name. For example, a stored procedure that looks for a person by their first name and last name, might look like this:

```
CREATE PROCEDURE [FindPerson] ( @qstr VARCHAR(32) )
AS
    SELECT *
    FROM [Person]
    WHERE fname like '%'+@qstr+'%' OR lname like '%'+@qstr+'%'
```

Obviously you can pass more than one parameter and name them anything you like. For example, a procedure to add a person might look like this:

```
CREATE PROCEDURE [CreatePerson]
    ( @fname VARCHAR(32), @lname VARCHAR(32), @email VARCHAR(64) )
AS
INSERT INTO [Person] ( [fname], [lname], [email] )
    VALUES ( @fname, @lname, @email )
RETURN @@IDENTITY
```

7.2 Quoting

There are many reserved words in T-SQL (or at least in Microsoft SQL Server), so the need to escape or quote strings becomes important. It is usually a good idea to quote table and column names, as well as procedure names and parameters. You quote things by putting it in the `[` and `]` brackets. For example, to create a person table, you'd do something like:

```
CREATE TABLE [Person] (
    [id] INT IDENTITY(1,1),
    [fname] VARCHAR(32),
    [lname] VARCHAR(32),
    [email] VARCHAR(64),
    PRIMARY KEY([id])
);
```

Obviously, the same applies to stored procedures.

7.3 Flow Control

T-SQL has the obvious flow control structures, which are only slightly different from the ones in PL/SQL. For example, and `IF` statement is just:

```
IF condition
    statement
```

Where condition is some logical condition, possibly involving variables or queries, and statement is some T-SQL, possibly involving `BEGIN` and `END` to include a whole block of statements. For example:

```
IF (SELECT COUNT(*) FROM [Person] WHERE [fname]='John') > 5
    PRINT 'There are more than 5 Johns in the database.'
```

There is also the usual `WHILE` loop:

```
WHILE (SELECT AVG(salary) FROM [Employee]) < 40000
    UPDATE [Employee] SET salary=salary * 0.05;
```

Which would raise everyone's salary by 5% until the average is over $40k. Again, you can use the `BEGIN` and `END` to use more than one statement, etc.

There are many other fancier things you can do, but I'll let you discover them on your own. The Microsoft SQL Server documentation is a great tool in figuring out how to do something.

8 Triggers

Triggers are simply *stored procedures* that are ran automatically by the database whenever some event (usually a table update) happens. We won't spend a great deal of time talking about *how to* write triggers, because if you know how to write stored procedures, you already know how to write triggers. What we will discuss is how to set them up to be executed whenever something happens to the table.

9 PL/SQL Triggers

Triggers are basically PL/SQL procedures that are associated with tables, and are called whenever a certain modification (event) occurs. The modification statements may include `INSERT`, `UPDATE`, and `DELETE`.

The general structure of triggers is:

```
CREATE [OR REPLACE]
TRIGGER trigger_name
BEFORE (or AFTER)
INSERT OR UPDATE [OF COLUMNS] OR DELETE
ON tablename
[FOR EACH ROW [WHEN (condition)]]
BEGIN
...
END;
```

The usual `CREATE OR REPLACE` we have already seen with procedures and functions... `TRIGGER` specifies just what type of object we are creating.

The `BEFORE` (or `AFTER`) in the trigger definition refers to when you want to run the trigger, either before the actual database modification (update, delete, insert) or after.

The list of various statements, `INSERT OR UPDATE [OF COLUMNS] OR DELETE` refers to statements that trigger this trigger. You can specify all three, or just one. Let's say you wanted the trigger to fire only when you do a delete; well, then you'd only specify a `DELETE` in the list.

On some table specifies that the trigger is associated with such table. As we shall see later, this does not necessarily has to be a *table*, but could also be a *view*.

There are several types of triggers; ones *for each row* and others *per statement*. For example, when you're doing an update, you can have a trigger fire once for each thing being updated (if you update 20 rows, the thing would fire 20 times), or you can have it fire just once per statement (if a single update statement is updating 20 rows, the trigger would fire just once). This is what that `FOR EACH ROW` in the trigger definition means.

The PL/SQL block (between `BEGIN` and `END`) is a usual code block where you can place PL/SQL commands. The only limitation is that you cannot use `COMMIT` (or `ROLLBACK`) for obvious reasons.

9.1 Permissions

Just like with procedures and functions, creating triggers requires certain privileges which are not part of the default privilege set. If you cannot create triggers from these notes because of permissions, you (or the admin) has to `GRANT CREATE TRIGGER` privilege on your username.

For example, to allow user 'alex' to create triggers, I may do something like this:

```
GRANT CREATE TRIGGER TO alex;
```

Note that if you are accessing a public Oracle server you must ask the admin to setup these things for you.

9.2 Sample Table to be Triggered

Before we begin playing with triggers, let's create a simple table with which we can experiment:

```
CREATE TABLE PERSON (
    ID   INT,
    NAME VARCHAR(30),
    DOB  DATE,
    PRIMARY KEY(ID)
);
```

The above creates a `PERSON` table with an `ID`, a `NAME` and a `DOB` columns (fields). Whatever triggers we define in these notes will relate to this table.

Also, let's not forget to setup: `SET SERVEROUTPUT ON;`

9.3 Before Insert Trigger

Let's start out our quest to learn triggers with the simplest case. We have nothing in the database (our `PERSON` table is empty). Before we insert any data, we'd like to perform some operation (let's say for logging purposes, or whatever). We write a trigger to fire *before* the insert takes place.

```
CREATE OR REPLACE
TRIGGER PERSON_INSERT_BEFORE
BEFORE
INSERT
ON PERSON
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('BEFORE INSERT OF ' || :NEW.NAME);
END;
```

Now let us test it out:

```
INSERT INTO PERSON(ID,NAME,DOB) VALUES (1,'JOHN DOE',SYSDATE);
```

The single `INSERT` statement fires the trigger. When we run it, we get the print out of `'BEFORE INSERT OF JOHN DOE'`. Ie:

```
SQL> INSERT INTO PERSON(ID,NAME,DOB) VALUES (1,'JOHN DOE',SYSDATE);
BEFORE INSERT OF JOHN DOE

1 row created.
```

9.4 After Insert Trigger

Can you guess what the trigger would look like that would fire `AFTER` the insert? Well?

```
CREATE OR REPLACE
TRIGGER PERSON_INSERT_AFTER
AFTER
```

```
INSERT
ON PERSON
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('AFTER INSERT OF ' || :NEW.NAME);
END;
```

And with our 2nd test `INSERT`:

```
INSERT INTO PERSON(ID,NAME,DOB) VALUES (2,'JANE DOE',SYSDATE);
```

For a total result of:

```
SQL> INSERT INTO PERSON(ID,NAME,DOB) VALUES (2,'JANE DOE',SYSDATE);
BEFORE INSERT OF JANE DOE
AFTER INSERT OF JANE DOE

1 row created.
```

Notice that *both* triggers have fired. One *before* the `INSERT` the other one *after*.

9.5 Before Update Statement Trigger

Now that we have some data in the table, we can create an update trigger, that would fire whenever someone tries to update any person (or persons).

```
CREATE OR REPLACE
TRIGGER PERSON_UPDATE_S_BEFORE
BEFORE UPDATE
ON PERSON
BEGIN
    DBMS_OUTPUT.PUT_LINE('BEFORE UPDATING SOME PERSON(S)');
END;
```

Now, let's run an update...

```
UPDATE PERSON SET DOB = SYSDATE;
```

Which produces the result:

```
SQL> UPDATE PERSON SET DOB = SYSDATE;
BEFORE UPDATING SOME PERSON(S)

2 rows updated.
```

Note that is says `2 rows updated` but we've only seen one `BEFORE UPDATING SOME PERSON(S)`, meaning that our trigger only fired once. This is because we did not specify `FOR EACH ROW` (which we'll do next).

Btw, from now on, we'll leave out a few details (I'll assume you can figure out how to write `PERSON_UPDATE_S_BEFORE` trigger, and such, etc.)

9.6 `FOR EACH ROW` Before Update Trigger

Right now, all we are doing is adding a `FOR EACH ROW` to last example:

```
CREATE OR REPLACE
TRIGGER PERSON_UPDATE_BEFORE
BEFORE UPDATE
ON PERSON
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('BEFORE UPDATING ' ||
       TO_CHAR(:OLD.DOB,'HH:MI:SS') || ' TO ' ||
       TO_CHAR(:NEW.DOB,'HH:MI:SS'));
END;
```

We're also printing out (displaying) the old value of `PERSON.DOB` and the new value. Now, let's run our update statement:

```
UPDATE PERSON SET DOB = SYSDATE;
```

Which gives the results:

```
SQL> UPDATE PERSON SET DOB = SYSDATE;
BEFORE UPDATING SOME PERSON(S)
BEFORE UPDATING 10:54:06 TO 11:10:01
BEFORE UPDATING 10:54:06 TO 11:10:01

2 rows updated.
```

Notice that we still get the firing of the initial 'non' per row trigger (that's the first one), then the `FOR EACH ROW` trigger fires, which actually does run for each row that is updated (in this case, twice).

9.7 Special `IF` statements

Inside the PL/SQL block of a trigger we can use if statements to determine what statement caused the firing of the trigger. These are generally of the form: `IF inserting THEN...` where besides "inserting" you can also use `updating` and `deleting`. An example would be something like:

```
CREATE OR REPLACE
TRIGGER PERSON_BIUD
BEFORE INSERT OR UPDATE OR DELETE ON PERSON
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('INSERTING PERSON: ' || :NEW.NAME);
    ELSIF UPDATING THEN
```

```
        DBMS_OUTPUT.PUT_LINE('UPDATING PERSON: ' ||
            :OLD.NAME || ' TO ' || :NEW.NAME);
    ELSIF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('DELETING PERSON: ' || :OLD.NAME);
    END IF;
END;
```

Notice that we only have one `TRIGGER`, and we are using `IF` statements to determine what statement invoked it, and display an appropriate message in various cases. For example, when we do an insert:

```
INSERT INTO PERSON(ID,NAME,DOB) VALUES (3,'SUPERMAN',TO_DATE('09/05/1950','MM/DD/YYYY'))
```

Then we get output like:

```
INSERTING PERSON: SUPERMAN
```

If we go ahead and modify that person:

```
UPDATE PERSON SET NAME = 'BATMAN' WHERE NAME = 'SUPERMAN';
```

Then we get an output like:

```
UPDATING PERSON: SUPERMAN TO BATMAN
```

And finally, if we go ahead and delete that person:

```
DELETE PERSON WHERE NAME = 'BATMAN';
```

Then we would get output like:

```
DELETING PERSON: BATMAN
```

Please note that you will have to run `SET SERVEROUTPUT ON;` in SQL*Plus order to see the output.

9.8 Working with Views

For our next example, we will need to create a view (of PERSON table):

```
CREATE OR REPLACE
VIEW PERSON_VIEW AS
SELECT NAME FROM PERSON;
```

Now, we know that updating (or inserting) into a view is kind of pointless; however, we can provide this functionality using a trigger! For example:

```
CREATE OR REPLACE
TRIGGER PERSON_VIEW_INSERT
INSTEAD OF INSERT ON PERSON_VIEW
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('INSERTING: ' || :NEW.NAME);

    -- we can also do
    -- INSERT INTO PERSON(ID,NAME,DOB) VALUES (N,:NEW.NAME,SYSDATE);
END;
```

When we do an insert statement on `PERSON_VIEW`:

```
INSERT INTO PERSON_VIEW(NAME) VALUES ('SUPERMAN');
```

Which produces the result:

```
INSERTING: SUPERMAN
```

So, what did just happen??? Did we insert a value into a view? No, not really. What we did was fire a trigger when someone tried to insert a value into a `VIEW`.

Now, as the comment in the code indicates, we can actually simulate the insertion statement (by inserting the value into the `PERSON` table ourselves).

9.9 Trigger Exceptions (introduction)

Triggers become part of the transaction of a statement, which implies that it causes (or raises) any exceptions (which we'll talk about later), the whole statement is rolled back. Think of an exception as a flag that is raised when an error occurs. Sometimes, an error (or exception) is raised for a good reason. For example, to prevent some action that improperly modifies the database. Let's say that our database should not allow anyone to modify their `DOB` (after the person is in the database, their `DOB` is assumed to be static). Anyway, we can create a trigger that would prevent us from updating the `DOB`:

```
CREATE OR REPLACE
TRIGGER PERSON_DOB
BEFORE UPDATE OF DOB ON PERSON
FOR EACH ROW
BEGIN
    RAISE_APPLICATION_ERROR(-20000,'CANNOT CHANGE DATE OF BIRTH');
END;
```

Notice the format of the trigger declaration. We explicitly specify that it will be called `BEFORE UPDATE OF DOB ON PERSON`.

The next thing you should notice is the procedure call `RAISE_APPLICATION_ERROR`, which accepts an error code, and an explanation string. This effectively halts our trigger execution, and raises an error, preventing our `DOB` from being modified. An error (exception) in a trigger stops the code from updating the `DOB`.

When we do the actual update for example:

```
UPDATE PERSON SET DOB = SYSDATE;
```

We end up with an error, that says we `CANNOT CHANGE DATE OF BIRTH`.

```
SQL> UPDATE PERSON SET DOB = SYSDATE;
UPDATE PERSON SET DOB = SYSDATE
       *
ERROR at line 1:
ORA-20000: CANNOT CHANGE DATE OF BIRTH
ORA-06512: at "PARTICLE.PERSON_DOB", line 2
ORA-04088: error during execution of trigger 'PARTICLE.PERSON_DOB'
```

You should also notice the error code of `ORA-20000`. This is our `-20000` parameter to `RAISE_APPLICATION_ERROR`.

9.10 Viewing Triggers

You can see all your user defined triggers by doing a select statement on `USER_TRIGGERS`. For example:

```
SELECT TRIGGER_NAME FROM USER_TRIGGERS;
```

Which produces the names of all triggers. You can also select more columns to get more detailed trigger information. You can do that at your own leisure, and explore it on your own.

9.11 Dropping Triggers

You can `DROP` triggers just like anything. The general format would be something like:

```
DROP TRIGGER trigger_name;
```

9.12 Altering Triggers

If a trigger seems to be getting in the way, and you don't want to drop it, just disable it for a little while, you can alter it to disable it. Note that this is not the same as dropping a trigger; after you drop a trigger, it is gone.

The general format of an alter would be something like this:

```
ALTER TRIGGER trigger_name [ENABLE|DISABLE];
```

For example, let's say that with all our troubles, we still need to modify the DOB of 'JOHN DOE'. We cannot do this since we have a trigger on that table that prevents just that! So, we can disable it...

```
ALTER TRIGGER PERSON_DOB DISABLE;
```

Now, we can go ahead and modify the DOB :-)

```
UPDATE PERSON SET DOB = SYSDATE WHERE NAME = 'JOHN DOE';
```

We can then re-ENABLE the trigger.

```
ALTER TRIGGER PERSON_DOB ENABLE;
```

If we then try to do the same type of modification, the trigger kicks and prevents us from modifying the `DOB`.

10 T-SQL Triggers

T-SQL triggers are pretty much just like the PL/SQL triggers. For example, to raise an error when someone tries to update/insert records into `Person` table, you'd create something like:

```
CREATE TRIGGER sometrigname
ON [Person]
FOR INSERT, UPDATE
AS RAISERROR (50009, 16, 10)
```

The format is similar enough to PL/SQL that it doesn't need explanation. Obviously, after the `AS`, you can have a `BEGIN` and continue that with a whole huge procedure.

i.e.: Everything that you can do in PL/SQL, you can also do in T-SQL, except with possibly slightly different format.