TEMA 10. INTRODUCCIÓN A PL/SQL.

1.Introducción....2
2.El bloque PL/SQL. 2
3.Entrada Salida por consola....3
4.Elementos del lenguaje....4
4.1.Delimitadores. 4
4.2.Identificadores. 4
4.3.Literales. 5
4.4.Comentarios. 5
5.Tipos de datos simples, variables y constantes. 5
5.1.Numéricos. 5
5.2.Alfanuméricos. 5
5.3.Grandes objetos. 6
5.4.Otros. 6
5.5.Utilización de %type y %rowtype....6
5.6.Variables y constantes. 6
5.7.Conversión de tipos. 7
5.8.Operadores y precedencia de operadores. 7
6.Estructuras de control. 7
6.1.Control condicional. 7
6.2.Control iterativo. 10
7.Manejo de errores. 11

1. INTRODUCCIÓN.

PL/SQL es un lenguaje procedimental estructurado en bloques que amplía la funcionalidad de SQL. Con PL/SQL podemos usar sentencias SQL para manipular datos y sentencias de control de flujo para procesar los datos. Por tanto, PL/SQL combina la potencia de SQL para la manipulación de datos, con la potencia de los lenguajes procedimentales para procesar los datos.

Aunque PL/SQL fue creado por Oracle, hoy día todos los gestores de bases de datos utilizan un lenguaje procedimental muy parecido al ideado por Oracle para poder programar las bases de datos.

Como veremos, en PL/SQL podemos definir variables, constantes, funciones, procedimientos, capturar errores en tiempo de ejecución, anidar cualquier número de bloques, etc. Además, por medio de PL/SQL programaremos los disparadores de nuestra base de datos, tarea que no podríamos hacer sólo con SQL.

Una de las grandes ventajas que nos ofrece PL/SQL es un mejor rendimiento en entornos de red cliente-servidor, ya que permite mandar bloques PL/SQL desde el cliente al servidor a través de la red, reduciendo de esta forma el tráfico y así no tener que mandar una a una las sentencias SQL correspondientes.

Como para cualquier otro lenguaje de programación, debemos conocer las reglas de sintaxis que podemos utilizar, los diferentes elementos de que consta, los tipos de datos de los que disponemos, las estructuras de control que nos ofrece (tanto iterativas como condicionales) y cómo se realiza el manejo de los errores.

2. EL BLOQUE PL/SQL.

Un bloque es la unidad básica en PL/SQL.

n bloque PL/SQL consta de tres zonas:

Declaraciones: definiciones de variables, constantes, cursores y excepciones.

Proceso: zona donde se realizará el proceso en sí, conteniendo las sentencias ejecutables.

Excepciones: zona de manejo de errores en tiempo de ejecución.

La sintaxis es la siguiente:

```
[DECLARE
        [Declaración de variables, constantes, cursores y excepciones]]
BEGIN [Sentencias ejecutables]
        [EXCEPTION Manejadores de excepciones]
END;
```

Los bloques PL/SQL pueden anidarse a cualquier nivel. Como hemos comentado anteriormente el ámbito y la visibilidad de las variables es la normal en un lenguaje procedimental. Por ejemplo, en el siguiente fragmento de código se declara la variable aux en ambos bloques, pero en el bloque anidado aux con valor igual a 10 actúa de variable global y aux con valor igual a 5 actúa como variable local, por lo que en la comparación evaluaría a FALSE, ya que al tener el mismo nombre la visibilidad dominante sería la de la variable local.

```
DECLARE
        aux number := 10;
BEGIN
        DECLARE
                aux number := 5;
        BEGIN
                ...
                IF aux = 10 THEN  ….
                ...
        END;
END;
```

3. ENTRADA SALIDA POR CONSOLA

Para mostrar un valor por pantalla:

```
DBMS_OUTPUT.PUT_LINE(cadena);
```

En caso de que el valor no sea una cadena, se puede usar la función TO_CHAR para transformarlo.

Para leer valores por pantalla se puede usar las variables de sustitución y/o el comando ACCEPT.

Variable de sustitución (&variable): pueden aparecer directamente en la sentencia SELECT sin necesidad de definirla, anteponiéndola el símbolo & y SQL nos preguntará el valor que queremos asignarle.

```
SET VERIFY OFF -- para no mostrar por pantalla el valor anterior de la variable
SET SERVEROUTPUT ON

DECLARE
 v_fecha DATE:='&fecha';
BEGIN
 DBMS_OUTPUT.PUT_LINE('La fecha introducida es: '||TO_CHAR(v_fecha)); --esta es la forma correcta
 DBMS_OUTPUT.PUT_LINE('La fecha introducida es: '||TO_CHAR('&fecha')); -- lee de nuevo la fecha
END;
```

ACCEPT permite declarar una variable de SQL y leer su valor poniendo un mensaje en el Prompt.

```
ACCEPT variable [NUMBER|CHAR|DATE] [FORMAT][PROMPT text] [HIDE]
```

Para utilizar la variable se accede a ella anteponiéndole el símbolo &.

No se puede utilizar ACCEPT para leer variables dentro de un bloque PL/SQL, si queremos utilizarlo debemos hacerlo fuera.

```
SET VERIFY OFF
SET SERVEROUTPUT ON
ACCEPT precio NUMBER FORMAT 9999 PROMPT 'Teclea precio: '
ACCEPT iva NUMBER FORMAT 99.99 PROMPT 'Teclea IVA: '
ACCEPT uni NUMBER FORMAT 999 PROMPT 'Teclea unidades: '
/*
 A partir de declare comienza el bloque PL
*/
DECLARE
 V_precio NUMBER:= &precio;
 V_iva NUMBER:= &iva;
 V_uni NUMBER:= &uni;
 factura number(9,3);
BEGIN
 Factura:= v_uni*v_precio + v_uni*v_precio* (v_iva/100);
Dbms_output.put_line(' El importe es : ' || to_char(factura));
END;
```

Para ver las variables creadas

Comandos útiles para la gestión de estas variables:

DEFINE: Despliega la lista de variables con los valores que tienen

DEFINE [nombreVariable]: Despliega el valor que tiene almacenada la variable si está definida

DEFINE [nombreVariable] = [Valor]: Asigna un valor a una variable, este comando crea la variable si es necesaria

UNDEFINE [nombreVariable]: Elimina la definición de la variable

También podemos leer variables de teclado escribiendo el prefijo **&nombre_variable**. La diferencia es que no se almacena en la base de datos, como las definidas con ACCEPT (estas duran la sesión).

4. ELEMENTOS DEL LENGUAJE.

PL/SQL es un lenguaje no sensible a las mayúsculas, por lo que será equivalente escribir en mayúsculas o minúsculas, excepto cuando hablemos de literales de tipo cadena o de tipo carácter.

Las unidades léxicas se pueden clasificar en: Delimitadores, Identificadores, Literales y Comentarios.

4.1. DELIMITADORES.

PL/SQL tiene un conjunto de símbolos denominados delimitadores utilizados para representar operaciones entre tipos de datos, delimitar comentarios, etc. En la siguiente tabla puedes ver un resumen de los mismos.

DELIMITADORES EN PL/SQL.			
Delimitadores Simples		**Delimitadores Compuestos**	
Símbolo	**Significado**	**Símbolo**	**Significado.**
+	Suma.	******	Exponenciación.
%	Indicador de atributo.	**<>**	Distinto.
.	Selector.	**!=**	Distinto.
/	División.	**<=**	Menor o igual.
(	Delimitador de lista.	**>=**	Mayor o igual.
)	Delimitador de lista.	**..**	Rango.
:	Variable host.	**||**	Concatenación.
,	Separador de elementos.	**<<**	Delimitador de etiquetas.
*****	Producto.	**>>**	Delimitador de etiquetas.
"	Delimitador de identificador acotado.	**--**	Comentario de una línea.
=	Igual relacional.	**/***	Comentario de varias líneas.
<	Menor.	***/**	Comentario de varias líneas.
>	Mayor.	**:=**	Asignación.
@	Indicador de acceso remoto.	**=>**	Selector de nombre de parámetro.
;	**Terminador de sentencias.**		
-	**Resta/negación.**		

4.2. IDENTIFICADORES.

Los identificadores en PL/SQL, como en cualquier otro lenguaje de programación, son utilizados para nombrar elementos de nuestros programas. A la hora de utilizar los identificadores debemos tener en cuenta los siguientes aspectos:

Un identificador es una letra seguida opcionalmente de letras, números, $, _, #.

No podemos utilizar como identificador una palabra reservada.

- Ejemplos válidos: X, A1, codigo_postal.
- Ejemplos no válidos: rock&roll, on/off.

En PL/SQL se pueden definir identificadores acotados, en los que podemos usar cualquier carácter con una longitud máxima de 30 y deben estar delimitados por ". Ejemplo: "X*Y".

En PL/SQL existen algunos identificadores predefinidos y que tienen un significado especial ya que nos permitirán dar sentido sintáctico a nuestros programas. Estos identificadores son las palabras reservadas y no las podemos utilizar como identificadores en nuestros programas. Ejemplo: IF, THEN, ELSE ...

Algunas palabras reservadas para PL/SQL no lo son para SQL, por lo que podríamos tener una tabla

con una columna llamada 'type' por ejemplo, que nos daría un error de compilación al referirnos a ella en PL/SQL. La solución sería acotarlos. SELECT "TYPE".

4.3. LITERALES.

Los literales se utilizan en las comparaciones de valores o para asignar valores concretos a los identificadores que actúan como variables o constantes. Para expresar estos literales tendremos en cuenta que:

Los literales numéricos se expresarán por medio de notación decimal o de notación exponencial.

Ejemplos: 234, +341, 2e3, -2E-3, 7.45, 8.1e3.

Los literales de tipo carácter y de tipo cadena se deben delimitar con unas comillas simples.

Los literales lógicos son TRUE y FALSE.

El literal NULL que expresa que una variable no tiene ningún valor asignado.

4.4. COMENTARIOS.

En los lenguajes de programación es muy conveniente utilizar comentarios en mitad del código. Los comentarios no tienen ningún efecto sobre el código, pero sí ayudan mucho al programador o la programadora a recordar qué se está intentando hacer en cada caso (más aún cuando el código es compartido entre varias personas que se dedican a mejorarlo o corregirlo).

En PL/SQL podemos utilizar dos tipos de comentarios:

Los comentarios de una línea se expresarán por medio del delimitador --.

```
a:=b; --asignación
```

Los comentarios de varias líneas se acotarán por medio de los delimitadores /* y */.

```
/* Primera línea de comentarios.

Segunda línea de comentarios. */
```

5. TIPOS DE DATOS SIMPLES, VARIABLES Y CONSTANTES.

En cualquier lenguaje de programación, las variables y las constantes tienen un tipo de dato asignado (bien sea explícitamente o implícitamente). Dependiendo del tipo de dato el lenguaje de programación sabrá la estructura que utilizará para su almacenamiento, las restricciones en los valores que puede aceptar, la precisión del mismo, etc.

En PL/SQL contamos con todos los **tipos de datos simples** utilizados en SQL y algunos más. En este apartado vamos a enumerar los más utilizados.

5.1. NUMÉRICOS.

`BINARY_INTEGER`: Tipo de dato numérico cuyo rango es de -2147483647 .. 2147483647. PL/SQL además define algunos subtipos de éste: `NATURAL`, `NATURALN`, `POSITIVE`, `POSITIVEN`, `SIGNTYPE`.

`NUMBER`: Tipo de dato numérico para almacenar números racionales. Podemos especificar su escala (-84 .. 127) y su precisión (1 .. 38). La escala indica cuándo se redondea y hacia dónde. Ejemplos. escala=2: 8.234 -> 8.23, escala=-3: 7689 -> 8000. PL/SQL también define algunos subtipos como: `DEC`, `DECIMAL`, `DOUBLE PRECISION`, `FLOAT`, `INTEGER`, `INT`, `NUMERIC`, `REAL`, `SMALLINT`.

`PLS_INTEGER`: Tipo de datos numérico cuyo rango es el mismo que el del tipo de dato `BINARY_INTEGER`, pero que su representación es distinta por lo que las operaciones aritméticas llevadas a cabo con los mismos serán más eficientes que en los dos casos anteriores.

5.2. ALFANUMÉRICOS.

`CHAR(n)`: Array de n caracteres, máximo 2000 bytes. Si no especificamos longitud sería 1.

`LONG`: Array de caracteres con un máximo de 32760 bytes.

`RAW`: Array de bytes con un número máximo de 2000.

`LONG RAW`: Array de bytes con un máximo de 32760.

`VARCHAR2`: Tipo de dato para almacenar cadenas de longitud variable con un máximo de 32760.

5.3. GRANDES OBJETOS.

`BFILE`: Puntero a un fichero del Sistema Operativo.

`BLOB`: Objeto binario con una capacidad de 4 GB.

`CLOB`: Objeto carácter con una capacidad de 2 GB.

5.4. OTROS.

`BOOLEAN`: TRUE/FALSE.

`DATE`: Tipo de dato para almacenar valores día/hora desde el 1 enero de 4712 a.c. hasta el 31 diciembre de 4712 d.c.

Hemos visto los tipos de datos simples más usuales. Los tipos de datos compuestos los dejaremos para posteriores apartados.

5.5. UTILIZACIÓN DE %TYPE Y %ROWTYPE

Hay ocasiones que las variables que usamos en PL/SQL se emplean para manipular datos almacenados en una tabla de la BD. En estos casos tendrá que ser la variable del mismo tipo que las columnas de las tablas.

Para esto se utiliza el atributo %TYPE:

```
DECLARE
 N_SALARIO EMPLE.SALARIO%TYPE;
 NOMBRE    EMPLE.APELLIDO%TYPE;
```

También es frecuente declarar registros con el mismo formato que las columnas de las tablas. Para ello se utiliza el atributo **%ROWTYPE**, similar a %TYPE. El registro tendrá los mismos campos y del mismo tipo que la fila de la tabla correspondiente de la BD.

```
DECLARE
 R_DEPART DEPART%ROWTYPE;
BEGIN
-- INICIALIZACIÓN DE CAMPOS DE LA VARIABLE
 R_DEPART.DEPT_NO:=44;
 R_DEPART.DNOMBRE:='CUARENTA Y 4';
 R_DEPART.LOC:='MADRID';
 INSERT INTO DEPART VALUES(R_DEPART.DEPT_NO,  R_DEPART.DNOMBRE, R_DEPART.LOC);
DBMS_OUTPUT.PUT_LINE('REGISTRO INSERTADO.');
END;
```

5.6. VARIABLES Y CONSTANTES.

Para declarar variables o constantes pondremos el nombre de la variable, seguido del tipo de datos y opcionalmente una asignación.

Si es una constante antepondremos la palabra `CONSTANT` al tipo de dato (lo que querrá decir que no podemos cambiar su valor).

Formato:

```
<Nombre_constante> CONSTANT <tipo> := <valor>;
PI constant real:=3.1415;
```

Podremos sustituir el operador de asignación en las declaraciones por la palabra reservada `DEFAULT`. También podremos forzar a que no sea nula utilizando la palabra `NOT NULL` después del tipo y antes de la asignación. Si restringimos una variable con `NOT NULL` deberemos asignarle un valor al declararla, de lo contrario PL/SQL lanzará la excepción `VALUE_ERROR` (más adelante veremos lo que son las excepciones, de momento diremos que es un error en tiempo de ejecución).

```
id SMALLINT;
hoy DATE := sysdate;
pi CONSTANT REAL:= 3.1415;
id SMALLINT NOT NULL; --ilegal, no está inicializada
id SMALLINT NOT NULL := 9999; -- correcto
```

El alcance y la visibilidad de las variables en PL/SQL será el típico de los lenguajes estructurados basados en bloques, aunque eso lo veremos más adelante.

5.7. CONVERSIÓN DE TIPOS.

Aunque en PL/SQL existe la conversión implícita de tipos para tipos parecidos, siempre es aconsejable utilizar la conversión explícita de tipos por medio de funciones de conversión (TO_CHAR, TO_DATE, TO_NUMBER, …) y así evitar resultados inesperados.

5.8. OPERADORES Y PRECEDENCIA DE OPERADORES.

Los operadores nos permiten relacionar dos operandos.

Tenemos los mismos tipos operadores que en SQL: Operadores aritméticos, lógicos y relacionales.

En PL/SQL se establece dicha precedencia para establecer un criterio a la hora de utilizar varios operadores. Si dos operadores tienen la misma precedencia lo aconsejable es utilizar los paréntesis para alterar la precedencia de los mismos ya que las operaciones encerradas entre paréntesis tienen mayor precedencia.

PRECEDENCIA DE OPERADORES.	
Operador.	**Operación.**
**, NOT	Exponenciación, negación lógica.
+, -	Identidad, negación.
*, /	Multiplicación, división.
+, -, ||	Suma, resta y concatenación.
=, i=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparaciones.
AND	Conjunción lógica
OR	Disyunción lógica.

6. ESTRUCTURAS DE CONTROL.

En la vida constantemente tenemos que tomar decisiones que hacen que llevemos a cabo unas acciones u otras dependiendo de unas circunstancias o repetir una serie de acciones un número dado de veces o hasta que se cumpla una condición. En PL/SQL también podemos imitar estas situaciones por medio de las estructuras de control que son sentencias que nos permiten manejar el flujo de control de nuestro programa, y éstas son dos: condicionales e iterativas.

6.1. CONTROL CONDICIONAL.

Las estructuras de control condicional nos permiten llevar a cabo una acción u otra dependiendo de una condición. Vemos sus diferentes variantes:

`IF-THEN`: Forma más simple de las sentencias de control condicional. Si la evaluación de la condición es `TRUE`, entonces se ejecuta la secuencia de sentencias encerradas entre el `THEN` y el final de la

sentencia.

SENTENCIA IF-THEN.	
Sintaxis.	**Ejemplo.**

IF condicion THEN
secuencia_de_sentencias;
END IF;
	`IF (b<>0) THEN`
	`c:=a/b;`
	`END IF;`

Veamos un ejemplo, vamos a leer un número de teclado y visualizar si es mayor de 100

```
DECLARE
  NUME NUMBER(5):=&NUMERO;
BEGIN
  IF NUME>100 THEN
     DBMS_OUTPUT.PUT_LINE('EL NÚMERO LEÍDO '||NUME ||' ES > DE 100');
  END IF;
END;
```

`IF-THEN-ELSE`: Con esta forma de la sentencia ejecutaremos la primera secuencia de sentencias si la condición evalúa a `TRUE` y en caso contrario ejecutaremos la segunda secuencia de sentencias.

SENTENCIA IF-THEN-ELSE.	
Sintaxis.	**Ejemplo.**

IF condicion THEN
Secuencia_de_sentencias1;
ELSE
Secuencia_de_sentencias2;
END IF;
	`IF (b<>0)THEN`
	`c:=a/b;`
	`else`
	`c:= a*b;`
	`END IF;`

`IF-THEN-ELSIF`: Con esta última forma de la sentencia condicional podemos hacer una selección múltiple. Si la evaluación de la condición 1 da `TRUE`, ejecutamos la secuencia de sentencias 1, sino evaluamos la condición 2. Si esta evalúa a `TRUE` ejecutamos la secuencia de sentencias 2 y así para todos los `ELSIF` que haya. El último `ELSE` es opcional y es por si no se cumple ninguna de las condiciones anteriores.

SENTENCIA IF-THEN-ELSIF.	
Sintaxis.	**Ejemplo.**

IF condicion1 THEN
Secuencia_de_sentencias1;
ELSIF condicion2 THEN
Secuencia_de_sentencias2; ...
[ELSE Secuencia_de_sentencias;]
END IF;
	`IF (operacion = 'SUMA') THEN resultado := arg1 + arg2;`
	`ELSIF`
	`(operacion = 'RESTA') THEN resultado := arg1 – arg2;`
	`ELSIF`
	`(operacion = 'PRODUCTO') THEN resultado := arg1 * arg2;`
	`ELSIF (arg2 <> 0) AND (operacion = 'DIVISION') THEN`
	`resultado := arg1 / arg2;`
	`ELSE`
	`RAISE operacion_no_permitida;`
	`END IF;`

Veamos un ejemplo, vamos a calcular una subida de salario de los empleados de tal forma que el sueldo debe estar entre 1 y 50€ y la subida se realizará de la siguiente forma:

Si es menor o igual a 10, la subida será 100€.

Si está entre 11 y 20, subida de 150€

Si está entre 21 y 30, subida de 175€

Si está entre 31 y 40, subida de 200€

Si es mayor de 40 la subida será 230€

Actualizar el salario de todos los empleados del departamento leído. Si el departamento no está entre 1 y 50, visualizar departamento erróneo y no se realizará ninguna actualización.

```
DECLARE
 DEP NUMBER(5) :=&DEPAART;
 SUBIDA NUMBER(5);
BEGIN

IF DEP BETWEEN 1 AND 50
THEN
     DBMS_OUTPUT.PUT_LINE('CORRECTO');
     IF DEP<=10 THEN SUBIDA := 100;
      ELSIF DEP BETWEEN 11 AND 20 THEN SUBIDA := 150;
      ELSIF DEP BETWEEN 21 AND 30 THEN SUBIDA := 175;
      ELSIF DEP BETWEEN 31 AND 40 THEN SUBIDA := 200;
      ELSE
         SUBIDA := 230;
     END IF;
     --ACTUALIZAMOS A LOS EMPLEADOS
     UPDATE EMPLE SET SALARIO = SALARIO + SUBIDA WHERE DEPT_NO = DEP;
     DBMS_OUTPUT.PUT_LINE('EMPLEADOS ACTUALIZADOS: '||SQL%ROWCOUNT);
ELSE
   DBMS_OUTPUT.PUT_LINE('DEPARTAMENTO ERRÓNEO: '||DEP);
  END IF;
END;
```

`CASE`: es una evolución en el control lógico. Se diferencia de las estructuras IF-THEN-ELSE en que se puede utilizar una estructura simple para realizar selecciones lógicas en una lista de valores.

SENTENCIA CASE	
Sintaxis.	**Ejemplo.**

CASE variable
WHEN expresión1 then valor1;
WHEN expresión2 then valor2;
WHEN expresión3 then valor3;
WHEN expresión4 then valor4;
ELSE valor5
END;
	`CASE ciudad`
	`WHEN 'MADRID' then dbms_output.put_line('Real Madrid');`
	`WHEN 'BARCELONA' then dbms_output.put_line('Barcelona FC');`
	`WHEN 'LACORUÑA' then dbms_output.put_line('Deportivo de la Coruña');`
	`ELSE dbms_output.put_line('SIN EQUIPO!!!');`
	`END CASE;`

CASE DE BÚSQUEDA. Cada cláusula WHEN puede tener su propia expresión a evaluar. En este caso, después del CASE no aparece ninguna expresión.

```
DECLARE
  PRECIO NUMBER(3):=100;
 DESCUENTO NUMBER(2):=0;
BEGIN
CASE
 WHEN PRECIO<11 THEN DESCUENTO:=2;
 WHEN PRECIO>10 AND PRECIO<25 THEN DESCUENTO:=5;
 WHEN PRECIO>24 THEN DESCUENTO:=10;
 ELSE DESCUENTO:=15;
END CASE;
DBMS_OUTPUT.PUT_LINE('DESCUENTO='||DESCUENTO);
END;
```

CASE DE COMPROBACIÓN. Calcula el resultado de la expresión que sigue a la cláusula CASE. En WHEN se comprueba si el valor de la expresión coincide con el que fija.

```
DECLARE
 PRECIO NUMBER(3):=100;
 DESCUENTO NUMBER(2):=0;
 CODIGO NUMBER:=1;
BEGIN
CASE CODIGO
```

```
 WHEN 1 THEN DESCUENTO:=20;
 WHEN 2 THEN DESCUENTO:=50;
 WHEN 3 THEN DESCUENTO:=30;
 ELSE DESCUENTO:=15;
END CASE;
PRECIO:=PRECIO-DESCUENTO;
DBMS_OUTPUT.PUT_LINE('DESCUENTO='||DESCUENTO);
DBMS_OUTPUT.PUT_LINE('PRECIO='||PRECIO);
END;
```

6.2. CONTROL ITERATIVO.

Estas estructuras nos permiten ejecutar una secuencia de sentencias un determinado número de veces.

`LOOP`: La forma más simple es el bucle infinito, cuya sintaxis es:

```
LOOP
        secuencia_de_sentencias;
END LOOP;
```

`EXIT`: Con esta sentencia forzamos a un bucle a terminar y pasa el control a la siguiente sentencia después del bucle. Un `EXIT` no fuerza la salida de un bloque PL/SQL, sólo la salida del bucle.

```
LOOP
        ...
        IF encontrado = TRUE THEN
                EXIT;
        END IF;
END LOOP;
```

`EXIT WHEN condicion`: Fuerza a salir del bucle cuando se cumple una determinada condición.

```
LOOP
        ...
        EXIT WHEN encontrado;
 END LOOP;
```

`WHILE LOOP`: Este tipo de bucle ejecuta la secuencia de sentencias mientras la condición sea cierta.

SENTENCIA WHILE LOOP.	
Sintaxis.	**Ejemplo.**

WHILE condicion LOOP
Secuencia_de_sentencias;
END LOOP;
	`WHILE (not encontrado)`
	`LOOP ...`
	`END LOOP;`

Veamos un ejemplo:

```
DECLARE
 CONTADOR BINARY_INTEGER:=0;
 NUM1 NUMBER(4):=&NUMERO1;
 NUM2 NUMBER(4):=&NUMERO2;
 SUMA NUMBER(5):=0;

BEGIN
IF NUM2 >= 0 THEN

  WHILE CONTADOR < NUM2 LOOP
   SUMA := SUMA + NUM1;
   CONTADOR :=CONTADOR + 1;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( 'LA SUMA DE ' || NUM1 ||  ' '|| NUM2  || ' VECES ES : '|| SUMA);
ELSE
   DBMS_OUTPUT.PUT_LINE( 'NO SE PUEDE SUMAR, NUM2  NEGATIVO: ' ||NUM2);
```

```
END IF;
END;
```

`FOR LOOP`: Este bucle itera mientras el contador se encuentre en el rango definido.

SENTENCIA FOR LOOP.	
Sintaxis.	**Ejemplo.**

FOR contador IN [REVERSE] limite_inferior..limite_superior LOOP Secuencia_de_sentencias; END LOOP;	`FOR i IN 1..3 LOOP --i=1, i=2, i=3 ... END LOOP;`
	
	`SELECT count(*) INTO num_agentes FROM agentes; FOR i IN 1..num_agentes LOOP ... END LOOP;`
	
	`FOR i IN REVERSE 1..3 LOOP --i=3, i=2, i=1 ... END LOOP;`

Veamos un ejemplo:

```
DECLARE
  NUM NUMBER(4):=&NUMERO;
BEGIN
DBMS_OUTPUT.PUT_LINE( 'LA TABLA DE ' || TO_CHAR(NUM));
FOR CONTADOR IN 1..10 LOOP
      DBMS_OUTPUT.PUT_LINE( TO_CHAR(NUM)||' * '|| TO_CHAR(CONTADOR)||' = '||
TO_CHAR(NUM*CONTADOR));
END LOOP;
END;
```

7. MANEJO DE ERRORES.

Muchas veces te habrá pasado que surgen situaciones inesperadas con las que no contabas y a las que tienes que hacer frente. Pues cuando programamos con PL/SQL pasa lo mismo, que a veces tenemos que manejar errores debidos a situaciones diversas. Vamos a ver cómo tratarlos.

Cualquier situación de error es llamada **excepción** en PL/SQL. Cuando se detecta un error, una excepción es lanzada, es decir, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones. La parte de manejo de excepciones es la parte etiquetada como `EXCEPTION` y constará de sentencias para el manejo de dichas excepciones, llamadas **manejadores de excepciones**.

EXCEPCIONES DEFINIDAS POR EL USUARIO	
Sintaxis.	**Ejemplo.**

WHEN nombre_excepcion THEN <sentencias para su manejo> .... WHEN OTHERS THEN <sentencias para su manejo>	`DECLARE`
	`  supervisor agentes%ROWTYPE;`
	`BEGIN`
	`  SELECT * INTO supervisor FROM agentes`
	`  WHERE categoria = 2 AND oficina = 3;`
	`  ...`
	`EXCEPTION`
	`  WHEN NO_DATA_FOUND THEN`
	`  --Manejamos el no haber encontrado datos`
	`WHEN OTHERS THEN`
	`  --Manejamos cualquier error inesperado`
	`END;`

La parte `OTHERS` captura cualquier excepción no capturada.

Las excepciones pueden estar definidas por el usuario o definidas internamente. Las excepciones predefinidas se lanzarán automáticamente asociadas a un error de Oracle. Las excepciones definidas por el usuario deberán definirse y lanzarse explícitamente.

EXCEPCIONES DEFINIDAS POR EL USUARIO	
Sintaxis.	**Ejemplo.**

DECLARE
nombre_excepcion EXCEPTION;
BEGIN
...
RAISE nombre_excepcion;
...
END;
	`DECLARE`
	`  categoria_erronea EXCEPTION;`
	`BEGIN`
	`  ...`
	`  IF categoria<0 OR categoria>3 THEN`
	`  RAISE categoria_erronea;`
	`  END IF;`
	`  ...`
	`EXCEPTION`
	`  WHEN categoria_erronea THEN`
	`  --manejamos la categoria errónea`
	`END;`

En PL/SQL podemos definir nuestras propias excepciones en la parte `DECLARE` de cualquier bloque. Estas excepciones podemos lanzarlas explícitamente con la sentencia `RAISE nombre_excepción`.

EXCEPCIONES PREDEFINIDAS EN ORACLE.		
Excepción.	**SQLCODE**	**Lanzada cuando ...**
ACCES_INTO_NULL	-6530	Intentamos asignar valor a atributos de objetos no inicializados.
COLECTION_IS_NULL	-6531	Intentamos asignar valor a elementos de colecciones no inicializadas, o acceder a métodos distintos de `EXISTS`.
CURSOR_ALREADY_OPEN	-6511	Intentamos abrir un cursor ya abierto.
DUP_VAL_ON_INDEX	-1	Índice único violado.
INVALID_CURSOR	-1001	Intentamos hacer una operación con un cursor que no está abierto.
INVALID_NUMBER	-1722	Conversión de cadena a número falla.
LOGIN_DENIED	-1403	El usuario y/o contraseña para conectarnos a Oracle no es válido.
NO_DATA_FOUND	+100	Una sentencia `SELECT` no devuelve valores, o intentamos acceder a un elemento borrado de una tabla anidada.
NOT_LOGGED_ON	-1012	No estamos conectados a Oracle.
PROGRAM_ERROR	-6501	Ha ocurrido un error interno en PL/SQL.
ROWTYPE_MISMATCH	-6504	Diferentes tipos en la asignación de 2 cursores.
STORAGE_ERROR	-6500	Memoria corrupta.
SUBSCRIPT_BEYOND_COUNT	-6533	El índice al que intentamos acceder en una colección sobrepasa su límite superior.
SUBSCRIPT_OUTSIDE_LIMIT	-6532	Intentamos acceder a un rango no válido dentro de una colección (-1 por ejemplo).
TIMEOUT_ON_RESOURCE	-51	Un timeout ocurre mientras Oracle espera por un recurso.
TOO_MANY_ROWS	-1422	Una sentencia `SELECT...INTO...` devuelve más de una fila.
VALUE_ERROR	-6502	Ocurre un error de conversión, aritmético, de truncado o de restricción de tamaño.
ZERO_DIVIDE	-1476	Intentamos dividir un número por 0.

Ahora que ya sabemos lo que son las excepciones, cómo capturarlas y manejarlas y cómo definir y lanzar las nuestras propias. Es la hora de comentar algunos detalles sobre el uso de las mismas.

El alcance de una excepción sigue las mismas reglas que el de una variable, por lo que si nosotros redefinimos una excepción que ya es global para el bloque, la definición local prevalecerá y no podremos capturar esa excepción a menos que el bloque en la que estaba definida esa excepción fuese un bloque nombrado, y podremos capturarla usando la sintaxis: **nombre_bloque.nombre_excepcion.**

Las excepciones predefinidas están definidas globalmente. No necesitamos (ni debemos) redefinir las excepciones predefinidas.

```
DECLARE
   no_data_found EXCEPTION;
BEGIN
   SELECT * INTO ...
EXCEPTION
   WHEN no_data_found THEN --captura la excepción local, no la global
END;
```

Cuando manejamos una excepción no podemos continuar por la siguiente sentencia a la que la lanzó.

```
DECLARE
    ...
BEGIN
  ...
  INSERT INTO familias VALUES (id_fam, nom_fam, NULL, oficina);
  INSERT INTO agentes VALUES (id_ag, nom_ag, login, password, 0, 0, id_fam, NULL);
  ...
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
         --manejamos la excepción debida a que el nombre de
         --la familia ya existe, pero no podemos continuar por
         --el INSERT INTO agentes, a no ser que lo pongamos
         --explícitamente en el manejador
END;
```

Pero sí podemos encerrar la sentencia dentro de un bloque, y ahí capturar las posibles excepciones, para continuar con las siguientes sentencias.

```
DECLARE
  id_fam NUMBER;
  nom_fam VARCHAR2(40);
  oficina NUMBER;
  id_ag NUMBER;
  nom_ag VARCHAR2(60);
  usuario VARCHAR2(20);
  clave VARCHAR2(20);
BEGIN
  ...
  BEGIN
         INSERT INTO familias VALUES (id_fam, nom_fam, NULL, oficina);
  EXCEPTION
         WHEN DUP_VAL_ON_INDEX THEN
                SELECT identificador INTO id_fam FROM familias WHERE nombre = nom_fam;
  END;
  INSERT INTO agentes VALUES (id_ag, nom_ag, login, password, 1, 1, id_fam, null);
  ...
END;
```

EJERCICIO:

Supongamos que queremos reintentar una transacción hasta que no nos dé ningún error. Para ello deberemos encapsular la transacción en un bloque y capturar en éste las posibles excepciones. El

bloque lo metemos en un bucle y así se reintentará la transacción hasta que sea posible llevarla a cabo.

SOLUCIÓN:

```
DECLARE
        id_fam NUMBER;
        nombre VARCHAR2(40);
        oficina NUMBER;
BEGIN
        ...
        LOOP BEGIN SAVEPOINT inicio;
                INSERT INTO familias VALUES (id_fam, nombre, NULL, oficina);
                ...
                COMMIT;
                EXIT;
                EXCEPTION WHEN DUP_VAL_ON_INDEX THEN ROLLBACK TO inicio; id_fam :=
                id_fam + 1;
                END;
        END LOOP;
        ...
END;
```

Continuemos viendo algunos detalles a tener en cuenta, relativos al uso de las excepciones.

✓ Cuando ejecutamos varias sentencias seguidas del mismo tipo y queremos capturar alguna posible excepción debida al tipo de sentencia, podemos encapsular cada sentencia en un bloque y manejar en cada bloque la excepción, o podemos utilizar una variable localizadora para saber qué sentencia ha sido la que ha lanzado la excepción (aunque de esta manera no podremos continuar por la siguiente sentencia).

```
DECLARE
    sentencia NUMBER := 0;
BEGIN
    ...
     SELECT * FROM agentes ... sentencia := 1;
     SELECT * FROM familias ... sentencia := 2;
    SELECT * FROM oficinas ...
     ...
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        IF sentencia = 0 THEN
          RAISE agente_no_encontrado;
        ELSIF sentencia = 1 THEN
          RAISE familia_no_encontrada;
        ELSIF sentencia = 2 THEN
          RAISE oficina_no_encontrada;
        END IF;
END;
```

Si la excepción es capturada por un manejador de excepción apropiado, ésta es tratada y posteriormente el control es devuelto al bloque superior. Si la excepción no es capturada y no existe bloque superior, el control se devolverá al entorno. También puede darse que la excepción sea manejada en un bloque superior a falta de manejadores para ella en los bloques internos, la excepción se propaga de un bloque al superior y así hasta que sea manejada o no queden bloques superiores con lo que el control se devuelve al entorno. Una excepción también puede ser relanzada en un manejador.

Oracle también permite que nosotros lancemos nuestros propios mensajes de error a las aplicaciones y asociarlos a un código de error que Oracle reserva, para no interferir con los demás códigos de error. Lo hacemos por medio del procedimiento:

```
RAISE_APPLICATION_ERROR(error_number, message [, (TRUE|FALSE)]);
```

Donde error_number es un entero negativo comprendido entre –20000..-20999 y message es una cadena que devolvemos a la aplicación. El tercer parámetro especifica si el error se coloca en la pila de errores (TRUE) o se vacía la pila y se coloca únicamente el nuestro (FALSE). Sólo podemos llamar a este procedimiento desde un subprograma. No hay excepciones predefinidas asociadas a todos los posibles errores de Oracle, por lo que nosotros podremos asociar excepciones definidas por nosotros a errores Oracle, por medio de la directiva al compilador (o pseudoinstrucción):

```
PRAGMA_INIT( nombre_excepcion, error_Oracle )
```

Donde nombre_excepción es el nombre de una excepción definida anteriormente, y error_Oracle es el número negativo asociado al error.

```
DECLARE
    no_null EXCEPTION;
    PRAGMA EXCEPTION_INIT(no_null, -1400);
    id familias.identificador%TYPE;
    nombre familias.nombre%TYPE;
BEGIN
     ...
    nombre := NULL;
    ...
    INSERT INTO familias VALUES (id, nombre, null, null);
EXCEPTION
    WHEN not_null THEN
    ...
 END;
```

Oracle asocia 2 funciones para comprobar la ejecución de cualquier sentencia. SQLCODE nos devuelve el código de error y SQLERRM devuelve el mensaje de error asociado. Si una sentencia es ejecutada correctamente, SQLCODE nos devuelve 0 y en caso contrario devolverá un número negativo asociado al error (excepto NO_DATA_FOUND que tiene asociado el +100).

```
DECLARE
    cod number;
    msg varchar2(100);
BEGIN
...
EXCEPTION
    WHEN OTHERS THEN
          cod := SQLCODE;
          msg := SUBSTR(SQLERRM, 1, 1000);
          INSERT INTO errores VALUES (cod, msg);
END;
```